Epidemic Store for Massive Scale Systems

Francisco António Ferraz Martins de Almeida Maia
Advisor: Prof. Rui Carlos Oliveira

April 30, 2015

DECLARAÇÃO

Nome: Francisco António Ferraz Martins de Almeida Maia
Endereço electrónico: fmaia@di.uminho.pt
Telefone: 964300393
Número do Bilhete de Identidade: 12129666
Título Tese: Epidemic Store for Massive Scale Systems
Orientador: Prof. Rui Carlos Oliveira
Ano de conclusão: 2015
Designação do Doutoramento: The MAP-I Doctoral Program Of The Universities of Minho, Aveiro and Porto

Universidade do Minho, 30/04/2015

Assinatura: _____

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, _____

Full name: _____

Signature: _____

Agradecimentos

Ao terminar esta etapa apercebo-me da aventura incrível que foram estes últimos anos. Tantas vezes a subir e contra-corrente mas sempre com chegadas surpreendentes e plenas. Chegadas essas que são impossíveis se estivermos sozinhos.

Agradeço ao Professor Rui Oliveira pelas viagens que fomos fazendo. Viagens na discussão de ideias, na exigência que me fez crescer, na sua capacidade de inspirar e por proporcionar um ambiente de trabalho único onde sempre me senti bem.

Agredeço a todo o HASLab e em particular ao GSD pelo ambiente fantástico criado entre todos. Em particular, agredeço ao Professor José Orlando Pereira pela sua incansável ajuda e paciência.

I would like to thank Etienne Rivière for his solicitous help and for the numerous and fruitful discussions. I also thank everyone in the University of Neuchâtel. It was a pleasure to get to know you all and share with you those weeks. In particular, I would like to thank Pascal Felber for the warm welcome and Anita Sobe for the one of a kind opportunity to check out the Montreux Jazz Festival.

Gracias a Enrique Armendariz por ayudarme en mis primeros pasos como investigador.

Obrigado a todos os que passaram pelo lab e OsSemEstatuto. Em particular ao Francisco Cruz, João Paulo, Miguel Matos, Ricardo Vilaça, Filipe Campos, Ana Nunes, Paulo Jesus, Jácome Cunha, Nuno Carvalho, Fábio Coelho, Nelson Gonçalves, Pedro Gomes, Ricardo Gonçalves, Miguel Borges, Nuno Castro e Tiago Jorge. Sem os momentos sérios e, sobretudo, sem os menos sérios convosco não seria possível este trabalho.

De um modo especial quero agradecer à minha família. Aos meus pais Rosalina e António, irmãos Luís e José e à minha irmã Margarida. Um obrigado que não cabe nas palavras por tudo aquilo que de palavras não precisa. Obrigado ainda aos meus avós, aos meus primos e tios porque com a sua vida tornam a família grande e verdadeira.

Um enorme obrigado aos meus amigos. Aqueles de sempre, aqueles que

são família, aqueles que estão sempre lá. Obrigado porque, de tantos modos diferentes, ajudaram a iluminar o caminho e a avançar.

Obrigado aos meus amigos de nova aventura / Gracias a mis amigos di nueva aventura: Maria e Fabinho, Antonella e Sebastien. Obrigado por tantos momentos verdadeiramente especiais / Gracias por tantos momentos verdaderamente especiales.

Grazie a Federica. Grazie per i nuovi colori con cui hai cambiato la mia vita.

Finalmente, algumas instituições apoiaram o trabalho apresentado nesta dissertação. A Fundação para a Ciência e Tecnologia (FCT) apoiou este trabalho através da bolsa de doutoramento (SFRH / BD / 71476 / 2010). O Departamento de Informática da Univesidade do Minho e o HASLab - High Assurance Software Lab. ofereceram-me as condições necessárias para o desenvolvimento deste trabalho.

Epidemic Store for Massive Scale Systems

Considering the state-of-the-art systems for data management, it is observable that they exhibit two main frailties when deployed in a large scale system. On one hand, coordination protocols used in traditional relational database management systems do not perform well when the system grows beyond tens of nodes. On the other hand, data management approaches that relax consistency guarantees, thus avoiding coordination, struggle with high levels of system churn.

In this dissertation, we present a completely decentralized, coordination-free, scalable and robust data store. Our design is aimed at environments with several thousands of nodes and high levels of churn. Offering the current ubiquitous key-value data structures and programming interfaces, we describe how to overcome challenges raised by the need to distribute data - essential for load balancing, to replicate data - the crux of fault tolerance, and to route requests - key to performability.

Alongside the design of our data store, we make several contributions in the context of distributed systems slicing. We propose a novel slicing protocol that overcomes state-of-the-art limitations. Additionally, we propose a novel epidemic algorithm for scalable and decentralized organization of system nodes into groups. This algorithm is used as an alternative to slicing at the core of our system. It organizes nodes into groups of parameterizable size without the need to have nodes knowing the system size. The contributions made on slicing protocols and the proposed group construction protocol are independent from the design of the data store. They are generic and can also be used as building blocks for other applications.

Armazenamento Epidémico para Sistemas de Larga Escala

Ao considerar o estado da arte no que diz respeito a gestão de dados, é possível observar que as soluções existentes exibem duas grandes fragilidades quando instaladas em sistemas de grande escala. Por um lado, os protocolos de coordenação utilizados nas bases de dados relacionais tradicionais não são capazes de gerir, de forma eficaz, mais de uma dezena de nós. Por outro lado, abordagens que relaxam a coerência dos dados evitando assim protocolos de coordenação, escalam melhor mas não conseguem lidar com elevados níveis de dinamismo. Nomeadamente, constante entrada e saída de nós do sistema.

Nesta dissertação apresentamos um sistema de armazenamento de dados completamente descentralizado, que não recorre a protocolos de coordenação, que é escalável e robusto. O desenho do nosso sistema visa ambientes com vários milhares de nós e elevados níveis de dinamismo. Oferecendo uma interface chave-valor, descrevemos como superar os desafios de distribuição de dados - essencial para balanceamento de carga, de replicação de dados - que permite tolerância a falhas e de direccionamento de pedidos - chave para o desempenho.

Além do desenho do nosso sistema de armazenamento fazemos várias contribuições no âmbito do fatiamento de sistemas distribuídos. Propomos um novo protocolo de fatiamento que resolve várias limitações das abordagens existentes. Adicionalmente, propomos um novo algoritmo epidémico para organizar um sistema em grupos de nós de forma descentralizada e escalável. Este algoritmo é utilizado como alternativa a protocolos de fatiamento no mecanismo interno do nosso sistema. Organiza os nós em grupos de tamanho parametrizável sem que os nós precisem de saber o tamanho do sistema. As contribuições feitas em algoritmos de fatiamento e o novo algoritmo de construção de grupos são independentes do sistema de armazenamento de dados. São genéricos e podem ser utilizados como componentes de outras aplicações.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Current times are characterized by the influence that information systems and the Internet have in society quotidian. Social networks and a variety of online services have been shaping and radically changing the way people, companies and governments interact. Associated with the change in paradigm came an exponential growth in the amount of digital data being produced and exchanged. According to [Gantz and Reinsel, 2011], "over the next decade, the number of servers (virtual and physical) worldwide will grow by a factor of 10, the amount of information managed by enterprise datacenters will grow by a factor of 50, and the number of files the datacenter will have to deal with will grow by a factor of 75, at least". Some of the most interesting computer science challenges of our time emerge from the need to deal with such massive amounts of data. In particular, from the need to store and manage it.

Processors are not getting faster as before, but instead it is easier to get hold of much larger sets in multi-core architectures [Sutter, 2005]. The move to multi-core architectures resembles that from centralized *supercomputers* to massive scale deployments of commodity hardware [Chang et al., 2008]. This is reinforced in the new Cloud Computing paradigm. Although the Cloud abstraction gives the illusion of a massive centralized computing environment, it is in fact supported by several data centers with thousands of machines. Furthermore, the ever increasing number of high performance, storage rich and always connected desktops and mobile devices leads us to distributed systems whose unprecedented scale demands new ways to manage and process data.

Designing and programming distributed systems is a particularly hard task [Kramer, 2007]. Scaling them is even harder as scale represents additional challenges. The increase in size is necessarily accompanied by the increase in failures and churn [Dean, 2009]. By churn we mean membership

dynamics, i.e., the movement of nodes entering and leaving the system. Notably, in systems of serveral thousands of machines and components, failures and churn become the rule instead of the exception. At those scales, studies have shown disk replacement rates up to 13% [Schroeder and Gibson, 2007] and RAM failures has high as 8% [Schroeder et al., 2009]. More importantly, failure rates grow with system size [Schroeder and Gibson, 2010]. Consequently, with systems continuously growing in size, traditional distributed system protocols often turn out to be inadequate to face these challenges. This observation is particularly true for data management systems.

Traditional relational database management systems are unable to cope with large scale scenarios. These database management systems struggle to scale out. In other words, after a certain point, adding more machines to the system does not increase the system's capacity or performance. This limitation is related to the CAP theorem [Gilbert and Lynch, 2002]. From this theorem follows that it is not possible to have a distributed system with high availability, strong consistency and partition tolerance at the same time. Instead, only two of these characteristics can coexist. Traditional RDBMS favor strong consistency and, in order to achieve strong consistency in the presence of network partitions, need coordination protocols. Coordination is costly, impairing the system's capacity to scale. In fact, they are known to scale only to a few dozen nodes [Gray et al., 1996; Lin et al., 2005].

Popular Internet services (eg. Facebook, Flickr, Twitter) face the need to handle massive amounts of data guaranteeing high availability. Since traditional relational database managements systems are unfitted for this task, other data management systems surfaced to cope with large scale data management. Examples are the key-value stores Dynamo [DeCandia et al., 2007], PNuts [Cooper et al., 2008], Bigtable [Chang et al., 2008] and Cassandra [Lakshman and Malik, 2010]. These systems are characterized by focusing on scalability and availability. These desirable properties stem from a new approach to data management. In contrast to traditional relational databases, these data stores do not offer atomic multi-item operations, which allow them to avoid inter-node coordination and hardly any kind of synchronization mechanisms. In particular, it is possible to take advantage of the lack of relationship between data in different nodes to scale the system balancing load through data partitioning. Such simplification, however, comes at the expense of a richer, more powerful, query language (eg. SQL). In fact, typically these data stores provide a simple *put* and *get* interface, and the computation of more complex operations (e.g. join operations) is executed at the client side. Although providing a simpler API and requiring extra work at the client side for certain operations, these new data stores are well suited for a large class of applications [Leavitt, 2010] and, notably, they are able to

scale to deployments of hundreds of nodes. However, these data stores are not able to perform well beyond those numbers. Most of these new data management systems are based on structured peer-to-peer protocols [DeCandia et al., 2007; Lakshman and Malik, 2010]. They typically use a distributed hash table (DHT) such as Chord [Stoica et al., 2003] or variants, to organize nodes and distribute data among them as well as routing and data discovery. These protocols assume a moderately stable system and current implementations of DHTs struggle with churn rates observable in real peer-to-peer systems [Rhea et al., 2004].

1.1 Problem Statement and Objectives

Considering the state-of-the-art systems for data management, it is observable that they exhibit two main frailties when deployed in a large scale system. On one hand, coordination protocols used in traditional relational database management systems do not perform well when the system grows beyond tens of nodes. On the other hand, data management approaches that relax consistency guarantees, thus avoiding coordination, struggle with high levels of system churn.

The problem we consider in this work is that of designing a completely decentralized, coordination-free, scalable and robust data store aimed at environments with several thousands of nodes and high levels of churn. That is, how to provide an usable system while avoiding coordination mechanisms that impair scalability and, at the same time, overcome state-of-the-art system limitations under churn.

Our goal is thus to devise a data store tailored for very large scale systems, offering the current ubiquitous key-value data structures and programming interfaces, while ensuring common guarantees with respect to fault tolerance and performability. This leads to major challenges on how to distribute data – essential for load balancing, how to replicate data – the crux of fault tolerance, and how to route requests – key to performability.

1.2 Contributions

Along the dissertation we present three main contributions. Firstly, we describe the design of a key-value store, DataFlasks, designed for massive scale systems. We built DataFlasks as a completely decentralized peer-to-peer system where all nodes play the same role. Every node runs the same set of algorithms and there is no hierarchy, structure or coordination of any kind. We designed DataFlasks based on a stack of unstructured proactive protocols. The objective was to build a data store with high scalability and resilience under highly dynamic environments.

Secondly, we make several contributions in the context of distributed systems slicing. Slicing a distributed system is the process of autonomously partitioning its nodes into k groups, named slices, based on some criteria. For instance, it is possible to slice the system according to node storage space or their uptime. Motivated by the fact that slicing can be used to implement one of the core components of DataFlasks, we propose a new slicing protocol Slead and one variant DSlead, that tackle several limitations of state-of-the-art slicing protocols. In particular, we address steadiness and high memory consumption issues as well as propose new features previously absent from slicing protocols. Among them we propose novel solutions to deal with non-uniform slices and to perform online and dynamic slice schema reconfiguration.

Thirdly, we propose a novel gossip-based algorithm for scalable and decentralized organization of system nodes into groups. This algorithm, is used as an alternative to slicing at the core of DataFlasks. It organizes nodes into groups of parameterizable size without the need to have nodes knowing the system size. The ability to define the group size clearly distinguishes this approach from that of slicing. Moreover, the protocol is designed to integrate with DataFlasks minimizing data transfer between nodes when group layout needs to be adjusted with respect to slicing protocols.

The contributions made on slicing protocols and the proposed group construction protocol are independent from the design of DataFlasks. They are generic and can also be used as building blocks for other applications.

1.3 Results

The work described in this dissertation was published in several international conferences. The following resulted directly from the work on DataFlasks:

- Slead: Low-Memory, Steady Distributed Systems Slicing.
  Francisco Maia, Miguel Matos, Etienne Rivière and Rui Oliveira.
  In the 12th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2012.

- Slicing as a Distributed Systems Primitive.
  Francisco Maia, Miguel Matos, Etienne Rivière and Rui Oliveira.
  In the Latin-American Symposium on Dependable Computing (LADC), 2013.

- DataFlasks: an Epidemic Dependable Key-Value Substrate.
  Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira and Etienne Rivière.
  In the International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments - Dependable Systems and Networks Workshops (DSN-W), 2013.

- DataFlasks: Epidemic Store for Massive Scale Systems.
  Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira and Etienne Rivière.
  In the 33rd IEEE Symposium on Reliable Distributed Systems (SRDS), 2014.

Additionally, the result of collaborations paving the way for this thesis or leveraging its research appear in the following publications:

- Scalable Transactions in the Cloud: Partitioning Revisited.
  Francisco Maia, José Enrique Armendáriz-Iñigo, M. Idoia Ruiz-Fuertes and Rui Oliveira.
  In the 12th International Symposium on Distributed Objects, Middleware, and Applications (DOA), 2010.

- About the Feasibility of Transactional Support in Cloud Computing.
  Francisco Maia, Rui Oliveira and José Enrique Armendriz-Iñigo.

In the Eighth European Dependable Computing Conference (EDCC), 2010.

- Worldwide Consensus.
  Francisco Maia, Miguel Matos, José Pereira and Rui Oliveira.
  In the 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2011.

- MeT: Workload Aware Elasticity for NoSQL.
  Francisco Cruz, Francisco Maia, João Paulo, Miguel Matos, Ricardo Vilaça, José Pereira and Rui Oliveira.
  In Eurosys, 2013.

- Autonomous Multi-dimensional Slicing for Large-Scale Distributed Systems.
  Mathieu Pasquet, Francisco Maia, Etienne Rivière and Valerio Schiavoni.
  In the 14th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), 2014.

- Workload-aware Table Splitting for NoSQL.
  Francisco Cruz, Francisco Maia, Rui Oliveira and Ricardo Vilaça.
  In the 29th Annual ACM Symposium on Applied Computing (SAC), 2014.

1.4 Outline

This dissertation is organized as follows. We begin by providing some background and context in Chapter 2 where we survey related work on distributed data management and peer-to-peer protocols. Then, we organize the core content of the dissertation into four chapters. Chapter 3 introduces DataFlasks describing its design and architecture. Besides providing the intuition behind how DataFlasks works, it also identifies the fundamental challenge to tackle for its implementation. More specifically, DataFlasks relies on a group construction component with demanding scalability and resilience requisites. Chapter 4 and Chapter 5 focus on two alternatives for

the implementation of such component. These two chapters are the more extensive ones and represent the two more significant contributions of this dissertation. They are strongly based on the four papers described previously as being direct results from our work on DataFlasks. Chapter 6 completes the main matter of the dissertation describing an experiment that serves as a proof of concept for the design of DataFlasks. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Background

This chapter focuses on providing some context for the work described in subsequent chapters. Considering that our goal is to build a scalable and robust data store we center the chapter in networked systems and distributed data management systems. We provide a brief historical overview that aims at going through the main motivations behind different approaches to distributed data management existing in the literature. We then focus on these state-of-the-art systems and describe the most important mechanisms that support them. Finally, as our data store follows a significantly different approach from these systems we provide some background on epidemic (or gossip-based) systems. These serve as the foundation for DataFlasks, which is built following an epidemic approach to data management.

2.1 Distributed data storage

From the beginning, computer system designers searched for ways to allow data sharing and collaboration [Satyanarayanan, 1990b]. While, originally, users were able to create and store files in a locally available file system, the advent of networked systems made it possible to provide access to files remotely stored in other machines in the network. Initially, remote file access was done recurring to actual file transfers over the network. This proved to be limited and the idea that remote files should appear to be local for users (*network transparency* and *location transparency*) drove the design of new approaches to remote file access.

Distributed file systems allow users to seamlessly access files that are not stored locally on their machines. In particular, users are given a common file system interface which they use as if it was a local file system and the file system does all the work underneath. An example of a notable distributed

file systems is the Andrew File System [Satyanarayanan, 1990a]. This file system relied on caching and by using *lock* and *unlock* operations at the node actually responsible for the file it ensured data consistency. However, systems such as Andrew File System suffer from limitations of availability. Failures in the system often render it unavailable. In order to tackle this limitation, the concept of *disconnected operations* emerged as a possible solution. Systems such as Ficus [Guy et al., 1990], Coda [Satyanarayanan, 1990a], Rumor [Guy et al., 1999] or Bayou [Terry et al., 1995] allow clients to perform operations over data even if they are temporarily disconnected from the network. Afterwards, conflict resolution mechanisms are attempted when the client becomes connected. These systems are mainly focused on providing services for mobility environments.

Although the concerns considered in the design of distributed file systems are considerably different from the type of concerns considered in this dissertation work, they are worth mentioning in order to understand how we arrived to our design and how it is an improvement considering the existing literature. In particular, systems such as Ficus and Rumor can be seen as primordial peer-to-peer systems were every replica of an object can be updated by every participating node hinting at a flat hierarchy of nodes. At the same time, the difficulty of providing shared access to objects while providing consistency is at the core of the great majority of data management system design. In fact, the eventual consistency concept proposed in the design of these distributed file systems is also revisited by more recent approaches to data management.

2.2 Data management systems

Alongside the evolution of distributed file systems also emerged the need to provide more efficient ways for programmers and applications to deal with data and files. In order to tackle such problem, database management systems were introduced. Moreover, with the proposal of the relational model [Codd, 1982], relational database management systems became the traditional (and probably still the most widespread) approach to data management.

Although relational database management systems have been the focus of intense research work and also subject to continuous commercial driven evolution, they exhibit some limitations. These systems are often deployed in a single machine. This has two significant problems: 1) if such machine fails the system becomes unavailable; 2) If the system becomes overloaded the only solution is a new, more powerful machine, which becomes very costly.

In order to solve problem 1), replication mechanisms have been implemented for these relational database management systems [Gray et al., 1996; Lin et al., 2005]. However, problem 2) remains unsolved as these mechanisms are known to scale rather poorly [Gray et al., 1996].

The reason that existing replication mechanisms struggle to scale is related with data consistency guarantees offered by relational database management systems. In fact, according to the CAP theorem [Gilbert and Lynch, 2002] it is not possible, for a distributed system, to provide both high availability and consistency in the presence of network partitions. Once network partitions are unavoidable, data management systems must choose to compromise either availability or consistency. Traditional relational database systems favor strong consistency compromising availability and, in order to achieve strong consistency in the presence of network partitions, these systems need coordination protocols. Coordination is costly and impairs the system's capacity to scale.

Nevertheless, considering web applications and the different kinds of online services available today, scalable data management systems are a necessity. This follows from the fact that not only there are unprecedented amounts of data to manage but also increasingly high number of requests to handle. Moreover, unavailability became commonly unacceptable as outages mean loss of sales or clients. This situation is exacerbated with the advent of the cloud computing paradigm. Cloud computing advertises scalable and always available systems. The paradigm proposes and abstraction level that masks the actual system infrastructure and gives the illusion of a scalable, always available, infinitely powerful system. Naturally, in order to build the cloud abstraction, the underlying infrastructure and software systems must be able to scale and be able to provide high availability,

In the pursuit of data management systems with high scalability and availability, a group of new data stores has been proposed. Among them are PAST [Druschel and Rowstron, 2001], Dynamo [DeCandia et al., 2007], PNuts [Cooper et al., 2008], Bigtable [Chang et al., 2008], Cassandra [Lakshman and Malik, 2010], Riak [Klophaus, 2010] and Infinispan [Marchioni, 2012]. Commonly known as NoSQL data stores, these data management systems offer relaxed consistency guarantees when compared with traditional relational database management systems. Relaxing consistency allows them to favor scalability and availability even in the presence of network partitions. Notably, these data stores can scale to deployments of hundreds of nodes.

Alongside the relaxed consistency guarantees, NoSQL data stores also provide a simpler interface and, as the name implies, do not support the richer query language SQL. Typically, they provide a simple key-value interface but each data store has its own data model and variants of the interface

providing, in some cases, additional functionality [Vilaça et al., 2010]. Although relaxing consistency guarantees and providing a simpler interface, these data stores proved to be successful and adequate to support a number of online services [Leavitt, 2010].

One of the key features of these data stores is how they implement data distribution and discovery. Leveraging scalability properties of peer-to-peer protocols, all these data stores rely on a distributed hash table such as Chord [Stoica et al., 2003], Tapestry [Zhao et al., 2006], Pastry [Rowstron and Druschel, 2001] or variants to distribute and locate data objects. The exceptions are Bigtable [Chang et al., 2008] and PNUTS [Cooper et al., 2008], which are centrally managed instead. As described previously in Section 2.3, distributed hash tables were introduced to solve scalability problems of previous peer-to-peer systems. In particular, distributed hash tables avoid flooding the network for node discovery or content search. This mechanism enables NoSQL data stores scalability.

A slightly different approach is taken by OceanStore [Kubiatowicz et al., 2000]. In OceanStore strong consistency is still attainable has it is built on a two tier architecture. When updates are requested, a small set of replicas run an agreement protocol in order to choose a total order for the updates (first tier), which are then disseminated to other replicas (second tier). Even so, similarly to the other data management systems mentioned in this Section, OceanStore also relies on a distributed hash table for data discovery.

DataFlasks, the system proposed in this dissertation, is closely related to the aforementioned data stores from two points of view. Firstly, similarly to the NoSQL data stores, DataFlasks is designed with a simple key-value interface and without support for strong consistency. Secondly, it also relies on a peer-to-peer protocol for data distribution and discovery. Nevertheless, it distinguishes itself by the use of unstructured peer-to-peer protocols where existing data stores recur to a structured approach, i.e. distributed hash tables. Moreover, DataFlasks is based on a specific family of peer-to-peer protocols known as *epidemic* protocols.

2.3 Peer-to-peer systems

Still considering data sharing, another branch of research was dedicated to peer-to-peer systems. As the name implies, in these systems, peers (or nodes) communicate directly with one another avoiding the need to use a centralized server. The idea behind this approach is achieving load balancing and, in some cases, privacy is also a core concern in peer-to-peer system design. Moreover, avoiding the centralized server and balancing the load among par-

ticipants allows peer-to-peer systems to scale. However, peers need to discover each other and building a scalable and efficient peer discovery mechanism is one of the main challenges of peer-to-peer computing [Stoica et al., 2003].

As mentioned, Ficus and Rumor could be considered peer-to-peer systems in their initial phases. However, the term only had significant impact with the advent of Napster [Napster] and Seti@Home [Anderson et al., 2002]. Even so, neither one of these two systems is a purely peer-to-peer system. In Seti@Home there is no direct contact among peers. Data is sent for processing at participating nodes and returned to centralized servers. In Napster, while data transfer was in fact done among participating peers, a centralized catalog of music was still needed.

The first openly available pure peer-to-peer system was probably Gnutella [Ripeanu, 2001]. Gnutella was fully distributed and nodes contacted each other directly. Nodes maintained references for other nodes and could issue requests directly to those nodes. However, discovering new peers in the network and searching for content was done recurring to flooding. This proved not to scale [Chawathe et al., 2003].

The fact that node discovery and request routing was a serious impairment to Gnutella's and Gnutella-like systems' scalability led to the design of a new family of distributed systems. These are commonly known as Distributed Hash Tables. Examples are CAN [Ratnasamy et al., 2001], Freenet [Clarke et al., 2001], Pastry [Rowstron and Druschel, 2001], Chord [Stoica et al., 2003], and Tapestry [Zhao et al., 2006]. Taking Chord as a representative example, these systems provide a *lookup* primitive that given a *key* allows to discover the node that is responsible for holding it. Nodes are organized in a ring and by using consistent hashing techniques are responsible for an evenly distributed set of keys. Interestingly, Chord provides such operation without requiring nodes to know every other node in the system. Nodes hold a table of references to a subset of nodes in the system and are able to answer lookup requests in *O(log N)* communication steps where N is the size of the system. Naturally, these tables must be maintained up to date. The process of maintaining the structure of a distributed hash table is costly when system dynamism is high [Rhea et al., 2004].

Distributed Hash Tables in all their variants are currently used in many of the state-of-the-art data management systems. They provide the core for systems like Dynamo [DeCandia et al., 2007], PNuts [Cooper et al., 2008] and Cassandra [Lakshman and Malik, 2010]. However, it is important to notice that these data stores typically use a specific DHT variation called 'one-hop' DHT [Gupta et al., 2004; Vilaça et al., 2010]. This variation allows faster lookups but requires complete membership knowledge, i.e., each node knows

about all other nodes in the system. These data stores distributed nature is a common characteristic to our own approach.

Among the different DHT implementations proposed, there is one in particular that stands out for its widespread use nowadays. Initially proposed in [Maymounkov and Mazieres, 2002], many implementations of the Kademlia protocol have been proposed. The Mainline DHT is one of these implementations and it is considered to be the one supporting the largest DHT deployment known [Wang and Kangasharju, 2013]. The Kademlia DHT, from our perspective, represents a different generation of DHT protocols. It is designed avoiding the lookup rigidity of Chord and similar approaches. Kademlia allows parallel lookup requests and, in contrast with previous approaches, each request can be sent to a range of nodes instead of having to obey a precise lookup structure. The success of Kademlia stems from this step towards unstructured lookup mechanisms, whose flexibility allows Kademlia to better handle large scale deployments.

2.4 Epidemic protocols

Epidemic, also known as gossip, communication due to its fully decentralized nature is specially suited for data dissemination in massive scale environments. Notably, in such systems any kind of global knowledge is unattainable. Relying on information that grows linearly with system size does not scale and, therefore, is impractical in such a scenario. As a consequence, large scale protocols must solely depend on partial information about the system and on node-local decisions. Additionally, with very large number of nodes, faults and churn become the rule, not the exception. A system designed to be deployed in a massive scale scenario needs to handle faults and churn by design. Fortunately, *epidemic* or *gossip-based* protocols meet such requirements. Epidemic or Gossip-based protocols are known for their scalability and resilience under highly dynamic environments. They have been successfully used to build several webscale systems and services [Rivière and Voulgaris, 2011] like overlay construction and maintenance [Ganesh et al., 2001; Voulgaris et al., 2005a], consensus [Maia et al., 2011], data aggregation [Jesus et al., 2010], distributed slicing [Fernández et al., 2007; Gramoli et al., 2008; Maia et al., 2013a, 2012] and live video streaming [Matos et al., 2014].

To the best of our knowledge, in 1987, Demers et al. introduced the first mechanisms for replica maintenance based on the theory of epidemics [Bailey et al., 1975]. Three mechanisms were introduced: direct mail, anti-entropy and rumor mongering. Direct mail consists of sending a message with a

certain data update to all participating nodes. Naturally, this is not reliable as nodes may be disconnected at the time of the update. Anti-entropy works as a *repair* mechanism. Each node periodically contacts another node in the network and by exchanging information about their current state converge to the same state. In particular, this can mean exchanging missing updates and data objects. Finally, for rumor mongering each node sends the same message periodically to randomly chosen nodes in the system. It does so until a significant number of nodes reports to have already seen such message. Depending on the number of times a message is sent this mechanism can also lead messages to reach only a portion of the systems nodes.

These initial epidemic protocols assumed that each node could know every other node in the system and choose randomly among them the node to contact. Such approach does not scale. In subsequent systems, such as *pbcast* [Eugster et al., 2003], the notion of partial views was introduced allowing these systems to overcome the global knowledge limitation. Moreover, properties of epidemics allow to provide some guarantees with high probability even with partial views [Eugster et al., 2004].

Along this dissertation we will focus on epidemic protocols that rely only on partial views of the system. We consider a typical epidemic protocol that operates as follows. Each node knows a dynamic set of neighbors, called its *view*. The protocol progresses by having each node periodically exchanging knowledge with one or several of its neighbors.

Naturally, it is necessary to maintain the neighbor list refreshed as nodes can become disconnected or simply leave or enter the system. Besides maintaining the *view* refreshed it is also important to ensure it exhibits an important property. Epidemic protocols benefit from views composed by a uniformly random sample of nodes [Voulgaris and Steen, 2005]. If the *view* is, in fact, a random sample of nodes, choosing a random peer from such list is equivalent to choose randomly from all the nodes in the system, which is important for protocol convergence [Voulgaris and van Steen, 2013].

The problem of providing random views of nodes in the system falls in a well studied problem in large scale distributed systems that has been addressed by a family of protocols known as the *Peer Sampling Service*. These protocols are gossip protocols themselves. In a nutshell, each node keeps a set of nodes it knows. Periodically, it refreshes such set by contacting one ore more of those nodes and exchanging information. Notably, this apparently simple approach allows these protocols to provide each node with a random stream of uniformly sampled nodes. Examples of these protocols are Cyclon[Voulgaris et al., 2005a] and Newscast [Voulgaris et al., 2005b].

Interestingly, the collection of views generated by the Peer Sampling Service not only serves as the support for other epidemic protocols but

also as an information dissemination medium. From early work on random graphs [Erdős and Rényi, 1976], we know that it is possible, with arbitrarily high probability, to effectively disseminate data in an epidemic fashion provided that each node relays a sufficient number of messages. In particular, taking N as the number of nodes, each node must relay $ln(N)+c$ messages to have a probability of atomic *infection* of $p_{atomic} = \epsilon^{-\epsilon^{-c}}$, a value that quickly gets close to 1. For example for $c = 5$ the probability of atomic infection is $p_{atomic} = 0.993$. Considering *views* of size $ln(N) + c$, uniformly sampled from the all set of nodes, an overlay emerges that allows for epidemic data dissemination. Note that these views do not grow linearly with system size and are, therefore, a scalable dissemination mechanism.

The distinct characteristics of epidemic protocols make them ideal to the scenario we are considering in this dissertation. These will be the starting point of our design and DataFlasks will be entirely based on this type of protocols.

A specific class of epidemic protocols that are key to this thesis is that of slicing, more precisely distributed systems slicing. Slicing a large-scale distributed system is the process of autonomously partitioning its nodes into k groups, named *slices*. Slicing is achieved by grouping nodes with respect to some node-specific criteria, such as available storage, uptime, or bandwidth. Each slice corresponds to the nodes between two quantiles in a virtual ranking according to the criteria.

For instance, a system can be split in three groups, one with nodes with the lowest uptimes, one with nodes with the highest uptimes, and one in the middle. Such a partitioning can be used by applications to assign different tasks to different groups of nodes, e.g., assigning critical tasks to the more stable nodes and less critical tasks to other slices.

To the best of our knowledge there are two main slicing protocols available: Ranking [Fernández et al., 2007] and Sliver [Gramoli et al., 2008]. In this dissertation we extend these protocols by tackling some of their frailties and propose a new one. Both contributions are described in Section 4.

Chapter 3

DataFlasks: design and architecture

In this chapter we describe the design and architecture of DataFlasks. We start by focusing on the ideas that motivate our design and provide an overview of how DataFlasks works. We continue by describing the model considered, presenting the assumptions on which DataFlasks is built. Finally, we materialize our design in a concrete architecture describing each one of its components and how they are implemented.

3.1 Design

The goal that drove the work on DataFlasks was building a data store tailored for systems with thousands of nodes. The inherent characteristics of the systems with such scale presented challenges to tackle and motivated a number of design decisions. In fact, targeting large scale systems necessarily demands the ability to handle high dynamism. Nodes continuously enter and leave the system (*churn*) and components keep failing. DataFlasks is able to successfully handle such demanding environments while scaling effortlessly.

In order to provide compatibility with existing data store systems DataFlasks is a key-value data store. Client applications can write data via a *put* operation and retrieve it with a *get* operation. The *put* operation receives as input a key, an object and a version of the object. Each stored object may be retrieved through a *get* operation that receives the object key and desired version as input. The triple (key, object, version) is assumed to be unique but not enforced by DataFlasks. Evidently, various versions of each object are possible for a single key. However, two different write operations for the

same (key,version) pair may lead to inconsistency. In our design, we assume that the consistency of writes is handled outside the data store by the client middleware or application.

The pivotal idea guiding the design of DataFlasks is decentralization. In DataFlasks each node is autonomous and all nodes play the same role. A node progresses relying solely on local decisions without depending on any other node and on any kind of hierarchy. When a client issues a request to DataFlasks, such request is disseminated throughout the system and each node decides how to handle it. When a *get* is received, if the node holds the corresponding triple (key,object,version) it replies to the client. Otherwise, it ignores the request. In the case of a *put* operation, the node locally decides to store the data or to discard it. The decision to store or not the data is used to implement data distribution and replication. DataFlasks is designed in such a way that prevents all nodes to take the same decisions, which would lead to a system where all nodes either store every object or none at all. Both situations are undesirable as the former prevents data distribution and the latter defeats the purpose of the data store. At the same time, DataFlasks ensures that a sufficient number of nodes actually decides to store each data object in order to guarantee data replication.

In DataFlasks, the set of nodes that takes the same decisions on whether to store data objects or not is viewed as a group. Accordingly, we reduce the decision of which data to store to the decision of which group a node belongs to. Once that decision is made, each node is responsible for a subset of the data according to a deterministic mapping between the pair (key,version) of an object and the group it belongs to. Data is thus distributed by groups, providing load balancing, and replicated a number of times equal to the size of the group. Strikingly, in DataFlasks, each node is able to decide to which group it belongs without requiring any kind of coordination.

We have built DataFlasks using epidemic protocols (Section 2.4). In particular, unstructured and pro-active epidemic protocols. They are characterized by their independence from any kind of structure or hierarchy among nodes and by the fact that they rely on pro-active mechanisms for fault tolerance. Instead of explicitly detecting failures and act accordingly, pro-active protocols are continuously taking the initiative, being able to anticipate system repair. The choice for unstructured, pro-active protocols contrasts with the approach taken by current data stores. State-of-the-art data stores, as described in Chapter 2, often rely on a structured overlay network, typically a Distributed Hash Table (DHT), for request routing and data distribution. The DHT builds a node structure that allows requests to be quickly routed to the nodes responsible for handling them. Since DHTs are decentralized, they enable current data stores to scale better than previous centralized ap-

proaches. However, DHTs struggle with high churn rates. Each time a node enters or leaves the system the DHT structure is invalidated and must be repaired in order to accommodate the membership change. As request routing depends on the existence and correctness of the DHT structure, the structure repairing mechanism is very costly. This reactive recovery mechanism has been shown to significantly impair DHT performance under churn [Rhea et al., 2004].

By relying on epidemic protocols, DataFlasks is completely decentralized and coordination-free. Characteristics that make DataFlasks inherently scalable and able to cope with unprecedented levels of system dynamism, may it be caused by membership instability or by failures.

3.2 Model

Throughout the design of the system we considered the following system model. We target very large distributed systems, in the order of thousands of nodes. Each node has a unique identifier and nodes can communicate with nodes whose identifier they know. The system is asynchronous in that no assumption is made regarding the time a node can take to execute its algorithm or messages may take to reach their recipients. Nodes make use of local clocks and progress depends on the overall system timeliness.

We assume that nodes can enter or leave the system by their own initiative. Nodes can fail by crashing and may recover, but do not deviate from their algorithm. Nodes that recover may lose their states but no data corruption may occur. Moreover, it is accepted that messages can be lost or duplicated, but cannot be tampered with. However, communication channels are fair in that each message that is sent has a non-zero probability of being delivered.

3.3 Architecture and Implementation

Since DataFlasks is completely decentralized and all nodes are autonomous, the architecture of DataFlasks rests on that of the individual nodes. An overview of the architecture is depicted in Figure 3.1. The two most important mechanisms to consider are how requests are disseminated and how nodes determine the group they belong to. These mechanisms are abstracted in two architecture components. Request dissemination is the responsibility of a component called *node communication* while a component called *group construction* is responsible for maintaining information about which group

the node belongs to. Both these components are implemented with the use of epidemic protocols.

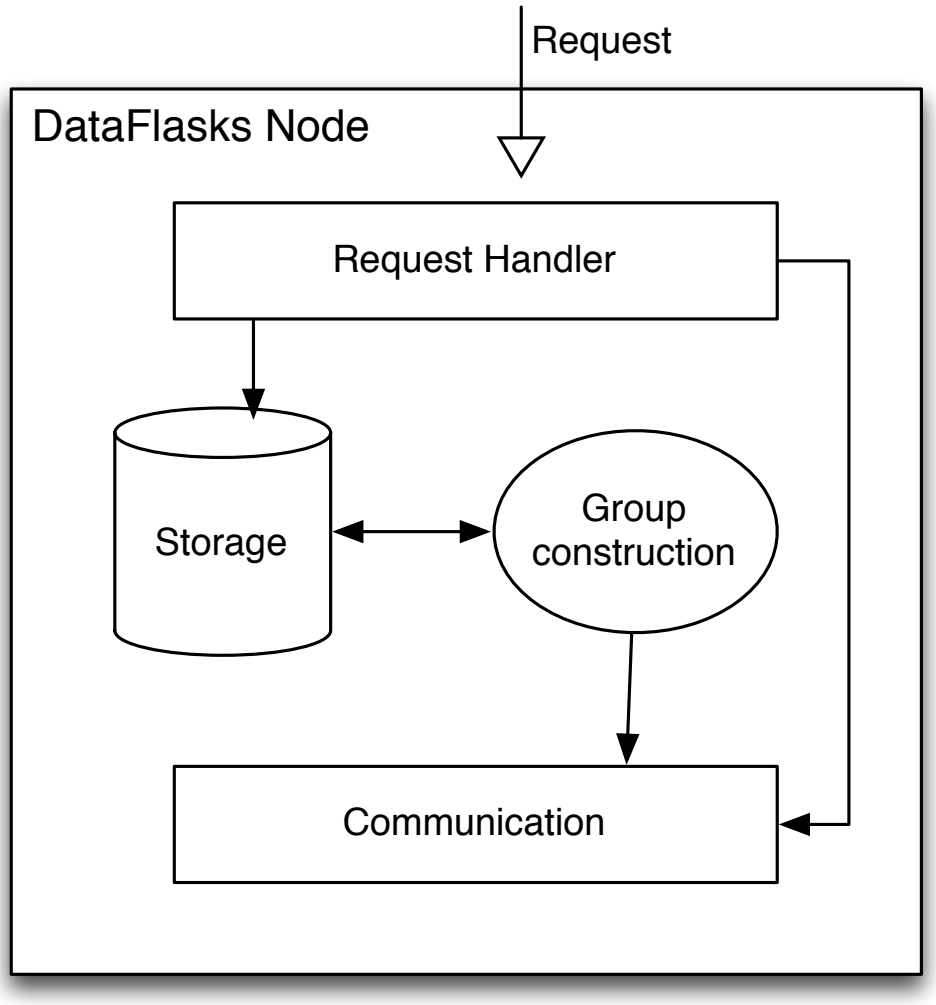


Figure 3.1: DataFlasks node architecture overview.

As described in Section 2.4, epidemic protocols progress by having nodes periodically exchanging messages with each other. In order to achieve this, nodes need to discover each other. This can be achieved with a specific class of epidemic protocols called Peer Sampling Service [Jelasity et al., 2007], which implements node discovery and membership maintenance. Additionally, on top of this type of service, it is possible to implement efficient data dissemination and extensive research exists on the matter. In particular,

suitable epidemic dissemination protocols exist [Demers et al., 1987; Eugster et al., 2003, 2004; Felber et al., 2012; Kermarrec and Van Steen, 2007; Kermarrec et al., 2003]. and can be directly used in DataFlasks. The main challenges in the implementation of DataFlasks appear in the context of the group construction component. To the best of our knowledge, there are no suitable epidemic protocols than can be used as an implementation of the group construction component. However, there is a class of epidemic protocols, which implement distributed systems slicing, that provided us with a starting point (Section 2.4). Currently, there are a number of existing slicing protocols available [Fernández et al., 2007; Gramoli et al., 2009; Montresor et al., 2008]. These protocols rely on a Peer Sampling Service (2.4) and notify each node of the slice (group) it belongs to. Although slicing protocols are able to construct groups they exhibit a number of limitations that hinder their immediate application to DataFlasks. In particular, these limitations are group instability and excessive memory consumption. In this dissertation, we improve on state-of-the-art protocols tackling their limitations with a novel slicing protocol. This work is the focus of Chapter 4.

Our slicing protocol can be applied in the context of DataFlasks group construction. It is scalable, able to cope with high levels of churn and also available for use outside the data storage context as a building block for other applications. However, it still exhibits two important limitations. On one hand, it is not possible for the user to specify a group size. The size of the group is always defined as a percentage of the system. Since the group size determines the data replication factor, it is important that such configuration parameter is available to the DataFlasks administrator. On the other hand, even after having drastically decreased instability rates of state-of-the-art slicing protocols, our protocol still allow nodes to change slice (group) frequently. This is highly undesirable because changing group leads to data transfers, which are costly.

In order to be able to tackle slicing protocols limitations we designed a novel group construction algorithm. This algorithm is the subject of Chapter 5. It is able to divide an arbitrarily large number of nodes into groups of user defined size. Moreover, it does so relying only on a Peer Sampling Service. The implementation of the group construction component, in both variants, represents the core contributions of the work on DataFlasks.

To complete the description of DataFlasks architecture it remains to describe two components: the *storage* component and the *request handling* component.

The storage component is responsible for the node-local data management. Using the the group identification maintained by the group construction component, the storage component can locally decide which client data

to store and which data to discard. Accordingly, the mapping of data to groups determines how data is distributed across the system. This mapping has two requisites. Firstly, data must be evenly distributed across groups in order to balance the load. Secondly, each node must be able to learn about such distribution solely based on the group it belongs to and on the number of groups that exist in the system. Note that, knowing the number of groups in the system is achievable using the aforementioned group construction mechanisms. In slicing protocols the number of groups is predefined and in our group construction protocol it is provided by the protocol itself. Determining to which group a certain key belongs is achieved by the use of an hash function. The hash function maps object keys with an arbitrary range size into keys of a fixed range size, doing so as evenly as possible over the target range. Using this target range data is distributed and balanced throughout the various groups and every node can locally determine if a certain key-value pair belongs to its data store. The storage component also abstracts the medium to which data is saved, which may vary for convenience.

Finally, the request handling component is the system's entrance point and is responsible for handling every request made to a DataFlasks node. Each time a request arrives, this component routes it through the workflow needed for processing. In particular, it delivers the request to the storage component for local processing and asks the node communication component to disseminate it to the other nodes.

3.4 Discussion

In this chapter we described the design and architecture of DataFlasks and sketched its operation. We proposed the use of epidemic protocols to implement two of its main components, the inter-node communication and group construction. Epidemic protocols are decentralized, rely on partial information of the system and eschew node coordination which are key characteristrics to deal with the massive scale systems envisioned in this work.

Epidemic communication is a topic that has been receiving a lot of attention from the research community. Several protocols that can be directly and advantageously used in DataFlasks have been proposed and used in production systems. One such protocol will be revisited in Chapter 6. In the next two chapters we focus on the challenges of group construction.

Chapter 4

Slicing for Data Distribution and Replication

4.1 Introduction

Large-scale systems are usually composed of highly heterogeneous nodes, according to their capacity, stability or any other application-specific requirements. The ability to distinguish between groups of nodes based on a discrete metric reflecting a criteria, allows to dynamically provide nodes to certain tasks according to their suitability. For instance, nodes with a higher uptime tend to be more stable for a given additional period than those with a small uptime. Partitioning the set of nodes into k several groups of increasing uptime, allows to assign critical services to more stable nodes, and less critical services to less stable ones. Examples include assigning privileged roles to more stable nodes to improve the quality of a streaming application [Wang et al., 2010], or allocating a data partition to a group of nodes in a key-value store [Maia et al., 2013b]. The operation of partitioning in k groups according to node-specific criteria is called distributed slicing [Fernández et al., 2007; Gramoli et al., 2009; Montresor et al., 2008].

In addition, these systems' scale turns any kind of global knowledge assumption unrealistic. In fact, any mechanism that relies on information that grows linearly with the system size is impractical. Large-scale systems require protocols designed to intrinsically scale to very large number of participating nodes, and consequently able to cope with highly dynamic environments. These requirements are addressed by a well studied class of protocols known as *epidemic*. These have been used previously to build several Internet-scale systems and services [Rivière and Voulgaris, 2011] like overlay construction and maintenance [Ganesh et al., 2001; Voulgaris et al.,

2005a], consensus [Maia et al., 2011] and data aggregation [Jesus et al., 2010]. Epidemic protocols have also been used in industrial systems such as Amazon's Dynamo [DeCandia et al., 2007] and Facebook's Cassandra [Lakshman and Malik, 2010].

There remain, however, several practical considerations that need to be taken into account to foster a broader adoption of epidemic-based systems for large-scale systems management and operation. Such disregard of practical aspects often stem from the use of simplifying models and simulations. This gap was observed previously even in fundamental primitives such as consensus [Chandra et al., 2007; Maia et al., 2011]. We consider that the same happens with *slicing*.

Slicing is the process of organizing a group of nodes into logical disjoint subgroups, called *slices*, according to some application dependent sortable criteria. Such logical division can be used for a variety of purposes such as the construction of hierarchical systems, identification of outliers, load-balancing or offering differentiated service levels [Gramoli et al., 2008]. Moreover, slicing is the natural candidate for managing heterogeneity which appears naturally in any large scale system from nodes with varying degrees of stability [Bhagwan et al., 2003] to different resource capacities [Hei et al., 2007; Saroiu et al., 2002]. As a matter of fact, popular systems such as the Skype VoIP service explicitly split the system into super and normal peers with different roles [Guha et al., 2006], and state-of-the-art video streaming systems like mTreeBone [Wang et al., 2010] offload most of the work to stabler nodes to improve streaming quality. We thus believe that there is a need for a distributed slicing primitive able to offer a generic but efficient slicing system that can be used by system and application designers.

Considering the design of DataFlasks, slicing can be used as the group construction component. It should able to group several thousands of nodes into groups in a completely decentralized and scalable way. Groups can be defined according to different criteria enabling customizable data distribution. For instance, using disk space as the sorting metric allows to reasonably adequate the amount of data each node should store. Moreover, slicing protocols robustness and continuous adaptation to change also enable data replication.

Unfortunately, despite the usefulness of slicing, state-of-the-art protocols still exhibit flaws that preclude, in our opinion, their immediate applicability as building blocks for large-scale applications. In particular, the applicability of slicing as the group construction component of DataFlasks. In this chapter we propose a new slicing algorithm called Slead and then a framework for slicing protocols. This framework can be instantiated with all the state-of-the-art slicing protocols and used to build new ones. We take advantage of such framework to further improve Slead, providing ad-

ditional features: multi-attribute slicing, non-uniform slice sizes, online slice reconfigurations and the ability to propose slicing as a substrate for other protocols by the provision of random set of nodes from the same slice, effectively implementing a slice-local *Peer Sampling Service* (PSS) [Jelasity et al., 2007].

In the design of SLEAD we begin by analyzing the state-of-the-art slicing protocols and propose improvements focusing on three previously disregarded metrics: *steadiness*, *slice variance* and *memory complexity*.

Steadiness is the ability of the protocol to take slice changes decisions only when necessary. It is the opposite of *slice instability*, measured by the distribution of the number of slice changes per second. A slice change can be legitimate, e.g., if the value of the nodes' attributes and thus the virtual ranking change, or if the size of the system changes. However, a slice change typically implies a considerable load for the overlying applications, as it requires reconfiguring the node for its new role, and often reconfiguring other nodes to take over its previous responsibilities. Undesired slice changes or oscillations between two slices tend to appear more frequently for nodes that lie at the "borders" of slices, that is, at the boundary of slices in the *virtual* ranking of all attributes. For instance, using slicing as a core component in our key-value store, a slice change results in discarding a potentially large fraction of hard state for the current slice and getting the new state from nodes of the new slice, which is costly.

Slice variance is a metric that reflects the correctness of the nodes allocation to slices, and in particular, the size distribution of the slices. It is important to notice that this metric significantly differs from the slice disorder metric used in previous work [Gramoli et al., 2009]. Slice variance does not distinguish whether a specific node is in the correct slice all the time but instead if the overall distribution of nodes into slices is close to the expected one, i.e., each slice is close in size to $\frac{N}{k}$ as possible (N is the size of the system). The *slice variance* is defined as the variance measured between the observed distribution of slices and $\frac{N}{k}$.

Thirdly, we consider the *memory complexity* imposed on nodes for deciding on their slice. This is a fundamental metric to assess scalability. A linear complexity requires keeping information in the order of the size of the system, and to maintain it through the system's dynamics, leading to poor performance and high costs.

We conducted experiments with two state-of-the-art protocols for distributed slicing [Fernández et al., 2007; Gramoli et al., 2008]. These protocols exhibit reasonable *slice variance* but suffer from serious *steadiness* and *memory complexity* problems. We address the two issues without impairing the original protocols performance w.r.t. other metrics. Our proposal, which we

named Slead, is a novel distributed slicing protocol whose design principles are generic enough to be adapted to other protocols such as [Fernández et al., 2007; Gramoli et al., 2008]. We address both issues with a *hysteresis* mechanism that significantly enhances *steadiness*. It is coupled with a bounded-memory state management mechanism based on Bloom filters [Bloom, 1970] that allows us to control *memory complexity* with a very limited impact on convergence and accuracy. Consequently, Slead is a better equipped slicing protocol for working as the group construction component of DataFlasks. The chapter is organized as follows. We begin analyzing current distributed slicing protocols in Section 4.2, highlighting their frailties. Section 4.3 is dedicated to our Slead protocol. In Section 4.4 we present the slicing framework and use it to propose several improvements to Slead. Section 4.5 concludes the chapter.

4.2 Analysis of state-of-the-art protocols

In this section we present, analyze and discuss two protocols, Ranking [Fernández et al., 2007] and Sliver [Gramoli et al., 2008] that to the best of our knowledge represent the state-of-the-art for distributed slicing. A complementary review and comparison of these protocols and other distributed slicing approaches can be found in [Gramoli et al., 2009].

Slicing is the process of organizing the set of nodes in a distributed system, into k groups called *slices*. Each slice must eventually be composed of the nodes that lie in the sequence of k subgroups ranked by increasing values of a sortable metric: if slices are $S_1, \ldots, S_i, \ldots, S_k$, then all nodes belonging to S_i must have a greater value for the metric than those in S_{i-1}, and a lower value for the metric than the nodes in S_{i+1}. Examples of sortable metrics include the uptime, available disk space, CPU or other application-specific metrics. In general, each node participating in a slicing protocol possesses an arbitrary local attribute and wishes to know the slice this value belongs to. The protocols work by performing pairwise exchanges of the local attribute with its neighbors. The decided slice may change after each such exchange, when the locally available information indicates that the local attribute value crosses a border in the global *virtual* ranking.

By assumption, each node in the system has access to a continuous stream of random nodes from the system. These nodes can be used as members of the node's view or to determine its position among the different slices. This is usually provided by an underlying proactive Peer Sampling Service (PSS) [Jelasity et al., 2007] that builds this stream of random nodes through a gossip-based periodic exchange of views between nodes. We also assume

that the number of slices, k, is known by all nodes. This value can easily be disseminated to all nodes through a gossip-based dissemination [Eugster et al., 2003], leveraging the PSS.

Ranking Ranking [Fernández et al., 2007], described by Algorithm 1, works in periodic cycles. It features an active and a passive thread. At each cycle, a node's *active thread* updates the local view by obtaining fresh random peers from the PSS. It then initiates an exchange with all these peers, simply sending its attribute (lines 7 to 10). Each contacted node processes the request with its *passive thread* (lines 11 to 27).

The principle of Ranking is to locally estimate the number of received attributes that are smaller than the receiver's. This allows estimating the position of the node's attribute in the *virtual* ranking, and decide on a slice (line 27). Ties in attribute values are disambiguated by comparing the node identifiers (line 16, second clause of the condition). Failure to do so by considering tied attributes on either the smaller or greater portion of the system would introduce estimation problems, particularly in scenarios where the attribute distribution is narrow (multiple nodes with the same attribute value).

As described, Ranking uses a sliding window mechanism by bounding the number of attributes considered and thus take churn (nodes' dynamics) into account.

Sliver Sliver [Gramoli et al., 2008], described by Algorithm 2, relies on the same basic idea of Ranking. Its fundamental difference though is to not only keep track of the attributes received but also to record their source nodes. Such apparently small difference has a significant impact and tackles a weakness in Ranking. Because the PSS is proactive and nodes periodically exchange the same information, eventually Ranking will consider the same attributes (providing from the same nodes) several times in the slice computation. If the underlying PSS does not provide completely uniform samples of the network, for instance due to heterogeneous network connections or to the nature of the shuffling operation used,[1] the biasing may strongly affect the accuracy of the slice estimation [Gramoli et al., 2009]. The longer the time slice considered, the more important is the bias introduced by selecting

[1] As demonstrated in [Jelasity et al., 2007] there is no such thing as a "perfect" peer sampling service; protocols that favor reactivity to take into account failed nodes usually impose a clustering ratio that is higher than that of a purely random network. It means that nodes in the vicinity of a given node are more likely to be seen twice in the flow of random nodes than what would have been the case with a purely random network.

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3      myAttribute ← ...
       // number slices, system parameter
4      k ← ...
       // list of latest collected attributes
5      attributeList ← ∅
       // current slice estimation
6      slice ← ⊥

7
   // active thread
8  every Δ sendAttribute()
9      view ← PSS.getView()
10     foreach p ∈ view
11         send myAttribute to p

12
   // passive thread
13 receive value from p
       // number of smaller attributes seen
14     smaller ← 0
       // total number of attributes seen
15     total ← 0
16     if attributeList.full then
17         attributeList.removeOlder()
18     if (value < myAttribute) ∨
19      (value == myAttribute ∧
20       p < myid) then
21         attributeList.add(true)
22     else
23         attributeList.add(false)
24     foreach a ∈ attributeList
25         if a then
26             smaller ← smaller + 1
27     total ← attributeList.size()
28     position ← smaller / total
29     slice ← k * position
```

Algorithm 1: Ranking [Fernández et al., 2007].

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3      myAttribute ← ...
       // number slices, system parameter
4      k ← ...
       // holds the received attributes and node ids
5      attributeList ← ∅
       // current slice estimation
6      slice ← ⊥

7
   // active thread
8  every Δ sendAttribute()
9      view ← PSS.getView()
10     foreach p ∈ view
11         send myAttribute to p

12
   // passive thread
13 receive value from p
       // number of smaller attributes seen
14     smaller ← 0
       // total number of attributes seen
15     total ← 0
16     if attributeList.contains(p,value) then
           // pair attribute and id become the head of list
17         attributeList.update(p,value)
18     else
19         if attributeList.full then
20             attributeList.removeOlder()
21             attributeList.add(p,value)
22         else
23             attributeList.add(p,value)
24     foreach a ∈ attributeList
25         if a.value < myAttribute then
26             smaller ← smaller + 1
27         else
28             if a.value == myAttribute ∧ a.id < myId then
29                 smaller ← smaller + 1
30     total ← attributeList.size()
31     position ← smaller / total
32     slice ← k * position
```

Algorithm 2: Sliver [Gramoli et al., 2008].

the same nodes several times. As Sliver keeps track of nodes identifiers, it is possible to overcome the impact of duplicates as well as provide a convergence proof as shown in [Gramoli et al., 2009]. Such a convergence proof is not applicable to Ranking.

Using a sliding window of observation Unfortunately, the continuous collection of attributes hinders scalability, as the memory required is proportional to the system size. This is the case for Ranking but is even more critical in Sliver as much more information is kept for each interaction. Due to this, both protocols bound memory usage by defining a *time to live* on attribute records, which enables to adjust memory consumption. In practice, defining a *time to live value* is equivalent to defining a maximum number of records each node can store. In our experiments this is the approach taken by keeping the records in a least-recently-used structure with custom size.

It is important to notice that the ability to forget records is crucial to cope with churn and changes in node local attribute values albeit with an impact on steadiness. In fact, defining a low value for the maximum amount of memory used allows the system to adapt to changes very fast but at the cost of unsteadiness, whereas increasing memory improves stability but slows the response to change.

Evaluation of Ranking and Sliver We now study the behavior of Ranking and Sliver with respect to *Steadiness* and *Slice variance*, for different amounts of memory consumption. The experiments were conducted with the help of the PeerSim simulation framework [Montresor and Jelasity, 2009] with a system size of 10 000 nodes and $k = 10$ slices with the event-based engine. For each experiment both protocols are stacked on top of the same PSS (Cyclon [Voulgaris et al., 2005a] in our case) and thus receive the same views enabling a direct comparison of results. As indicated in [Jelasity et al., 2007], Cyclon provides the best results of available PSS for the quality of the randomness of the streams of nodes constructed (in particular, low clustering ratios). This means we consider the best conditions for Ranking here; accuracy can only get worse as other PSS are considered. All presented results are the average of 10 executions. Due to the large number of points to plot, we applied a cubic spline transformation that summarizes plot data in order to improve readability. We consider the following configurations: Ranking and Sliver with memory size (maximum number of elements in *attributeList*) of 100, 1,000 and ∞.

For all configurations, the size of the view is 20. This means that the active thread of both Ranking and Sliver will contact 20 nodes with their

attribute value. If we consider the network formed by the PSS views to be random (a reasonable assumption in this case), each node will be on average contacted 20 times per cycle. Every time a node is contacted with an attribute value, its passive thread will integrate the received value and may decide on a slice change. In the worst case, a node may thus change its slice 20 times per cycle.

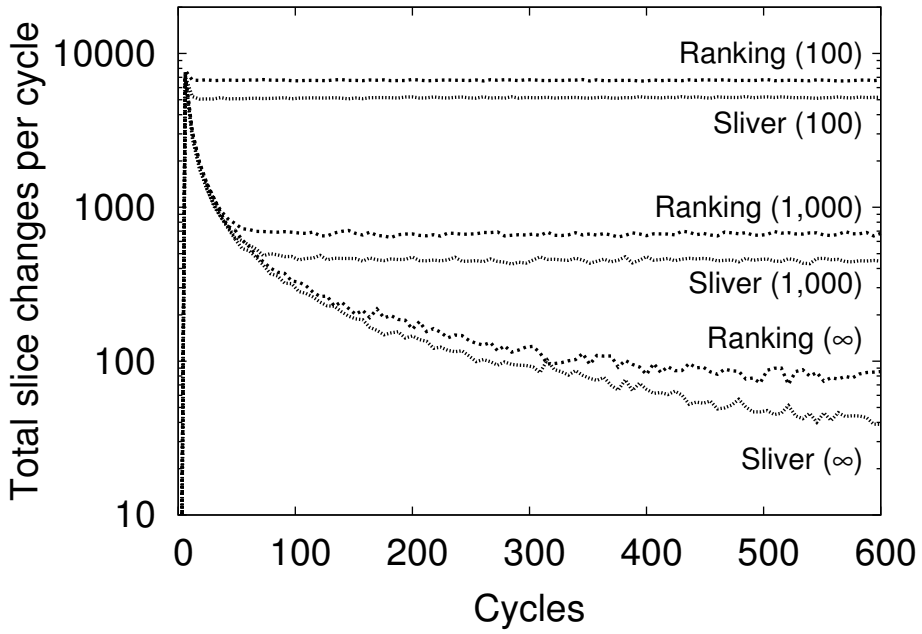


Figure 4.1: **Steadiness.** Evolution of the number of slice changes for 10,000 nodes and 10 slices over 600 cycles.

Figure 4.1 explores the *steadiness* of the various configurations. We represent the evolution of the number of changes per cycle, for all nodes (note the logarithmic scale for the y axis). As expected, due to the low number of values stored by both protocols, there is a major instability of the slice decisions in the beginning that result in a large number of slice changes, multiple times per cycle and per node.When using a bounded memory size, there is a *stabilization period* after which the number of slice changes per cycle remain almost constant. This stabilization period is the time it takes to fill the memory: 20 times 50 cycles makes for 1,000 entries in one case, 20 times 5 cycles makes for the 100 entries in the other. The number of slice changes, and thus *steadiness*, is thus directly linked to the memory size at each node.

Even a memory of *a tenth* of the total system size is synonym with ma-

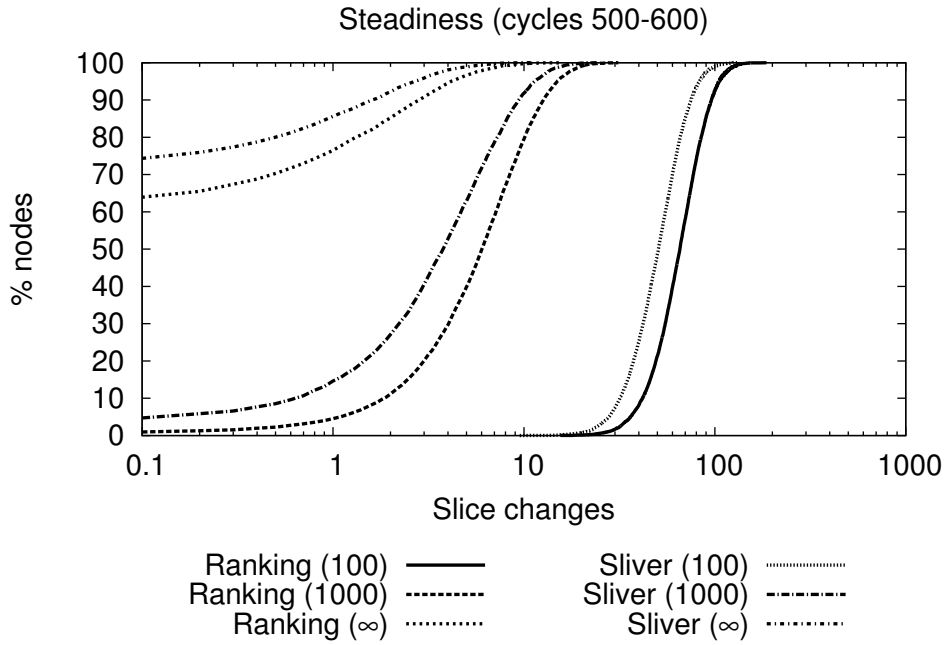


Figure 4.2: **Steadiness.** Cumulative changes over the last 100 cycles for 10,000 nodes and 10 slices.

jor slice attribution instability. Keeping system-size amount of information results in the protocols stabilizing, but very slowly. By cycle 600, Ranking will have seen 600 times 20 values, more than the size of the system, and still be unstable. As expected, Sliver is slightly more efficient for the same memory and stabilizes faster by discarding already known information and counting each attribute only once. Nonetheless, we do not see the stabilization of Sliver with a complete knowledge of the system as it would require much more than $\frac{10{,}000}{20} = 500$ cycles to get such a complete knowledge (latest missing attributes taking longer to be captured). We note that the difference between Ranking and Sliver would be higher if using a PSS yielding a lower-quality stream of nodes, e.g., where clustering would be more present.

Figure 4.2 presents the cumulative slice changes from cycle 500 to 600 which is enough for all configurations to stabilize. As expected, slice changes are not evenly distributed among all nodes and tends to affect nodes that are on, or next to, *slice borders* in the virtual ranking. In fact, even with knowledge of one *tenth* of the system (1 000 records), roughly 20% of the nodes change slices at least every 10 cycles. The result is deceptive for the usability of Ranking and Sliver in a real system as these nodes will be unusable or incur a heavy and persistent reconfiguration load on the system.

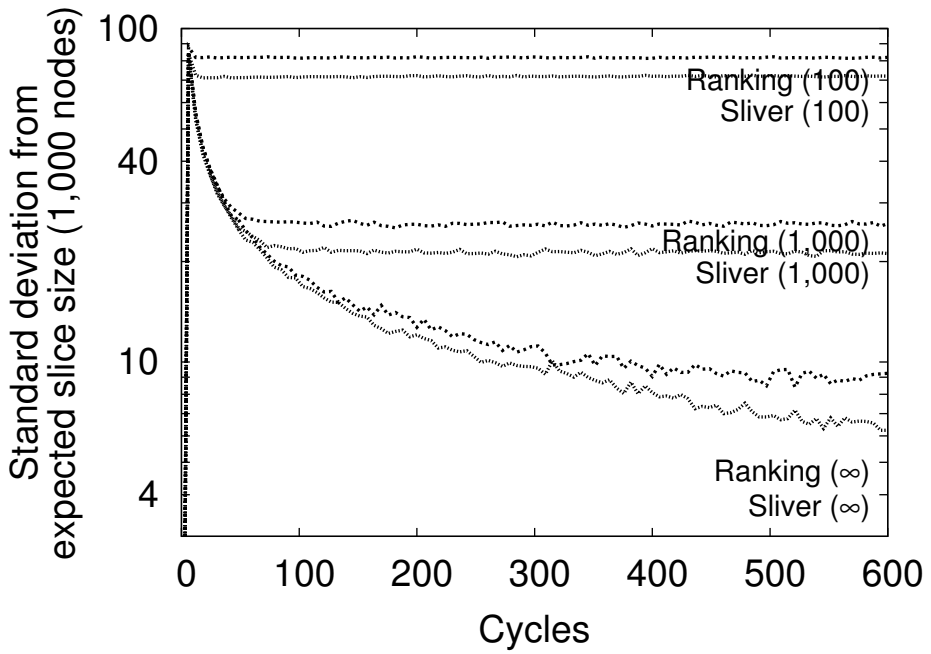


Figure 4.3: *Slice variance* Evolution of the slices std. dev. from 1,000 nodes for 10,000 nodes and 10 slices over 600 cycles.

Figure 4.3 presents the impact of the various configurations on *slice variance*. Here, we plot the standard deviation from the expected slice size (1,000 nodes). We observe that *slice variance* is heavily dependent on the memory used: more entries reduce the differences between slices while low memory (100 entries) results in an instability on slice size. Note that the distribution of slice sizes evolves over time: the large slices may be the smaller a few cycles later, due to the randomness in the slice attribution. This we attribute to the low memory available and resulting limited knowledge of the network.

Discussion These evaluations show that an immediate application of either protocol is problematic, particularly due to the *steadiness* problem, as a significant percentage of the system would be devoted to performing slice transitions without doing any useful work. These observations are the starting point and main motivation behind the solutions and protocol presented next.

4.3 Slead

In this Section we present SLEAD, a new distributed slicing protocol that addresses the problems of *steadiness* and *memory consumption* found in existing protocols and highlighted in the previous section.This is achieved without impacting *slice variance* (and thus the distance from an ideal slice distribution). In fact, SLEAD can achieve the same *slice variance* as previous protocols but with a significantly lower memory consumption as we demonstrate later in this Section. For the sake of clarity we introduce each mechanism independently which allows a better understanding of the impact of each of them.

Conceptually, SLEAD is similar to both Sliver and Ranking as in each cycle nodes send their local attributes to their neighbors and compute their position in the global ranking (and hence their slice) based on the attributes received in the recent past. The full pseudo-code of SLEAD is presented in Algorithm 3, and detailed and evaluated in the following sections.

4.3.1 Steadiness

Changing slice typically requires the node to change context and local state, which can be very expensive. As we have shown in Section 4.2, Sliver [Gramoli et al., 2008] and Ranking [Fernández et al., 2007] suffer from a *steadiness* problem in the slice estimation: a large fraction of nodes keep changing slices even in a stable network and long after bootstrap. In fact, this happens mainly because nodes close to the slice border are highly affected by small variations in their position estimation.

To address such fluctuations, we propose the use of a *hysteresis* mechanism that prevents such problematic changes. The basic idea is to look at the slice estimate over a period of time and only change slice if the slice proposal is done for a sufficient amount of rounds, or if the magnitude of the change is high enough. The number of rounds or the magnitude of the change needed is given by a parameter we call the *friction factor*.

The hysteresis component of SLEAD is presented in Algorithm 3, lines 20 to 24 and works as follows. At each cycle, the protocol computes the slice estimation (lines 18 to 20). The magnitude of the change is accumulated in a local variable, *current_difference*, which represents the cumulative difference between the current slice estimation and the one the protocol is suggesting as correct (line 21). As we compute the difference between the current slice and the estimated one, small fluctuations in the estimation are avoided since they do not go over the *friction factor* and thus *steadiness* is improved. If the estimated slice consistently points to a new value, the cumulative differ-

ence will eventually be greater than the *friction factor* and the protocol will effectively adopt the change to the new slice. Furthermore, as the hysteresis is based on cumulative differences the protocol is able to quickly adapt to abrupt changes in the system such as massive joins or failures. In fact, if the difference between the proposed slice and the current one is greater than the *friction factor*, the change will be immediate thus helping to effectively deal with dynamics.

Figures Figs. 4.4 to 4.6 present the impact of the hysteresis mechanism applied to Ranking and Sliver in the same scenario of Section 4.2 with *friction*=2. We only consider the versions with unbounded memory of both protocols as those achieve better results in both metrics as observed in Section 4.2. We observe that the hysteresis mechanism not only improves overall system *steadiness* (Figure 4.4) but also considerably reduces the amount of nodes that frequently changes slice (Figure 4.5, note that the x axis scale is logarithmic). Moreover, there is no impact on *slice variance* (Figure 4.6) meaning that despite avoiding unnecessary changes the protocols still converge to the optimal configuration when compared with their original versions.

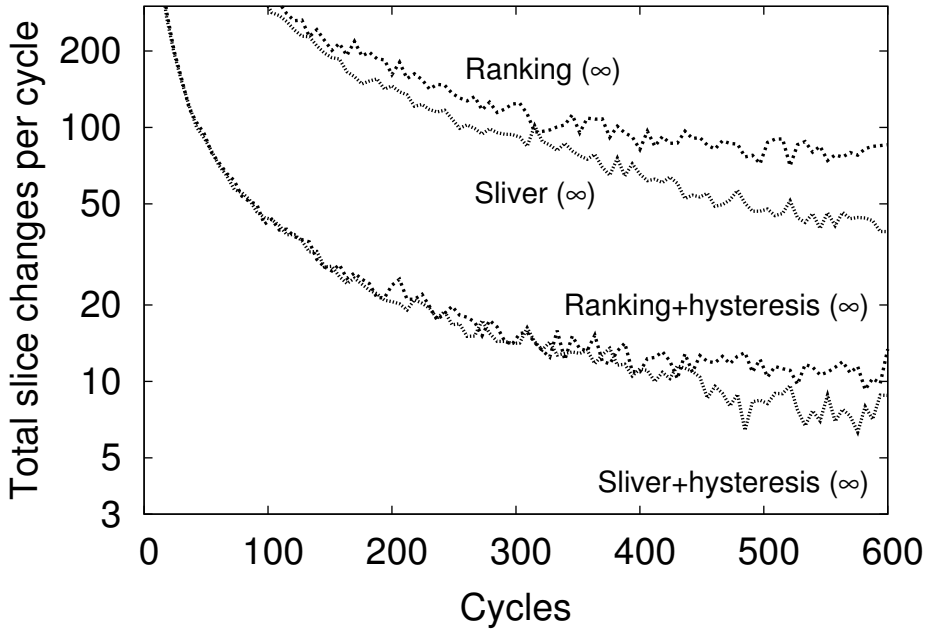


Figure 4.4: **Steadiness.** Evolution of the number of slice changes (10,000 nodes, 10 slices).

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3
4      myAttribute ← ...
       // number slices, system parameter
5      k ← ...
       // node identifiers whose attributes are smaller than the local one
6      smaller ← BloomFilter()
       // node identifiers whose attributes are greater than the local one
7      greater ← BloomFilter()
       // current slice estimation
8      slice ← ⊥
       // current value of cumulative changes attempts
9      current_difference ← 0
10
   // active thread
11 every Δ sendAttribute()
12     view ← PSS.getView()
13     foreach p ∈ view
14         send myAttribute to p
15
16 receive value from p
17     if (value < myAttribute ∨ (value == myAttribute ∧ p < myId)) then
18         smaller.add(p)
19     else
20         greater.add(p)
21     total ← smaller.size() + greater.size()
22     position ← smaller.size() / total
       // hysteresis mechanism
23     nextSlice ← k * position
24     current_difference ← current_difference + (slice − nextSlice)
25     if ‖current_difference‖ > friction then
26         slice ← nextSlice
27         myprotocol.current_difference ← 0
```

Algorithm 3: SLEAD protocol.

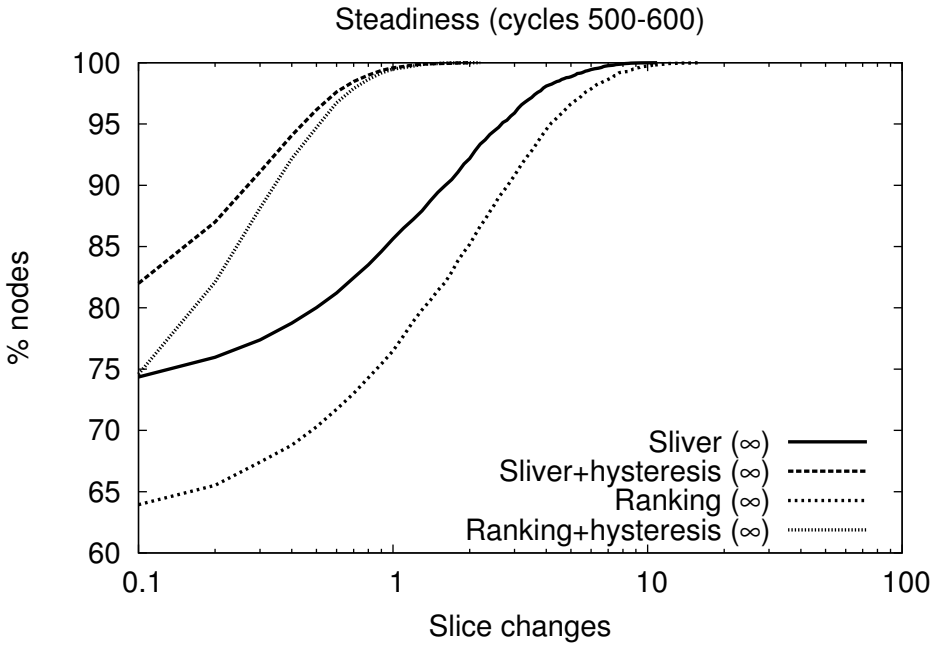


Figure 4.5: **Steadiness.** Cumulative changes over the last 100 cycles (10,000 nodes, 10 slices).

4.3.2 Memory usage

The other main frailty with existing slicing protocols is that the memory requirements depend on the system size and too low a memory impacts *slice variance* as observed in Figure 4.3. This is because Ranking and Sliver need to store the values of the attributes of other nodes (and the node *id* in the case of Sliver) to build adequate estimations of the slice position. The compromise taken in Sliver and Ranking is to use a least-recently-used structure that bounds memory consumption even though constraining estimation accuracy.

Our contribution to reducing memory usage rests on two key observations regarding the nature of distributed slicing. First, it is important to track which attributes (source nodes) have been considered in the past to avoid duplicates. Secondly, what really matters to the slice computation is not the values themselves but whether they are greater or smaller than the local attribute. The first observation directly calls for the use of a Bloom filter, a space-efficient data structure for tracking identifiers [Bloom, 1970][2].

[2]We note that using a Bloom filter can give false positives for the inclusion of an element in the set (here, a node identifier). However, the probability of a false positive for the identifier of a node with a greater attribute is the same as for a node with a smaller

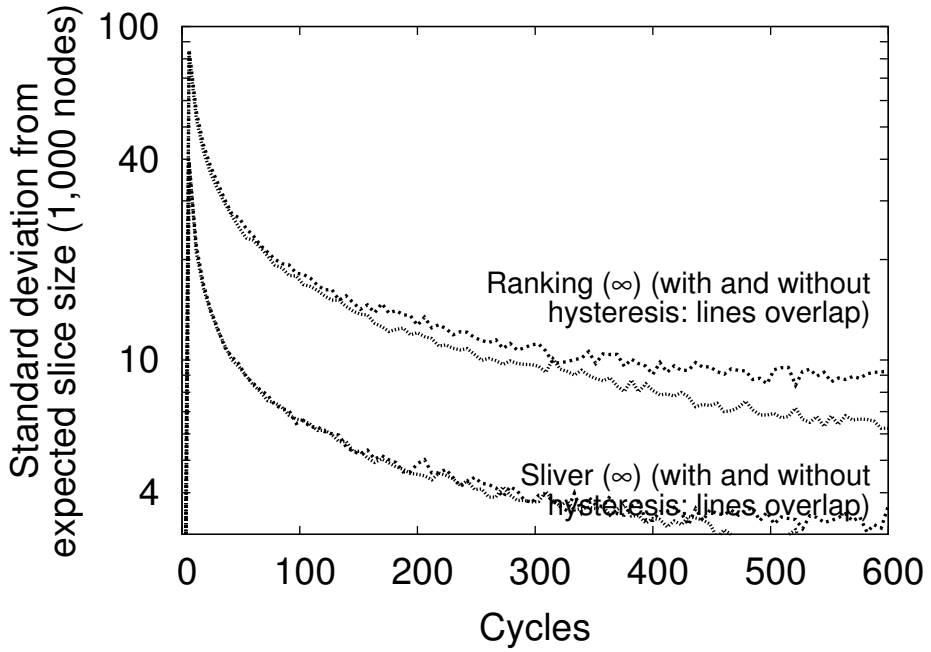


Figure 4.6: **Slice Variance.** Evolution of the slices std. dev. from 1,000 nodes (10,000 nodes, 10 slices).

The second one, leads to simply counting the greater and smaller observations, which only requires to keep two numbers instead of a list with all the occurrences.

Therefore, in Slead we use Bloom filters to track the node identifiers, which allows to track a significant higher number of ids using a bounded and small amount of memory. Assuming a pair `IP:port` as the node identifier (48 bits) and that attributes are encoded as long integers (64 bits), each entry requires 64 bits in Ranking and 112 in Sliver. For the memory configurations used previously with 100, 1000 and 10,000 entries (the unbounded version in practice corresponds to the system size), Ranking requires 6,400, 64,000 and 640,000 bits, whereas Sliver requires 11,200, 112,000 and 1,120,000 bits, respectively. On the other hand, a Bloom filter with a probability of false positives of 1×10^{-4} (the order of the system size) requires only 1,071, 10,899 and 109,158 bits for storing 100, 1,000 and 10,000 nodes respectively [Bloom, 1970], representing savings of around 90% when compared to Sliver. The next step is simply to count the number of elements in each Bloom filter and compute the slice estimation accordingly (lines 10 and 19). Please note that

attribute; henceforth the position estimation is not affected by such errors that are evenly spread on the attribute range space.

the addition to a Bloom filter is an idempotent operation and thus has no impact on the cardinality which can be easily computed from the filter fill ratio [Bloom, 1970].

To evaluate our mechanism, we compared Ranking and Sliver with unbounded memory which in practice corresponds to 640,000 and 1,120,000 bits respectively, and SLEAD with 218,316 bits which corresponds to the two Bloom filters with a capacity to store 10 000 node identifiers with a false positive probability of 1×10^{-4}. We detail the need for two bloom filters in the next section. To isolate the impact of the use of Bloom filters, SLEAD does not use the hysteresis mechanism in this experiment. The results are depicted in Figure Figs. 4.7 and 4.8 and as it is possible to observe despite using only 35% of Ranking's memory and 20% of Sliver's, SLEAD provides similar results for both *steadiness* and *slice variance*. Such memory improvements could be further increased by using more advanced Bloom filters that do not require setting an a priori filter size and are able to scale with the number of inserted elements [Almeida et al., 2007]. In fact, this benefits nodes that are on the low/high end of the attribute spectrum as they will not require significant memory for the smaller/larger Bloom filters.

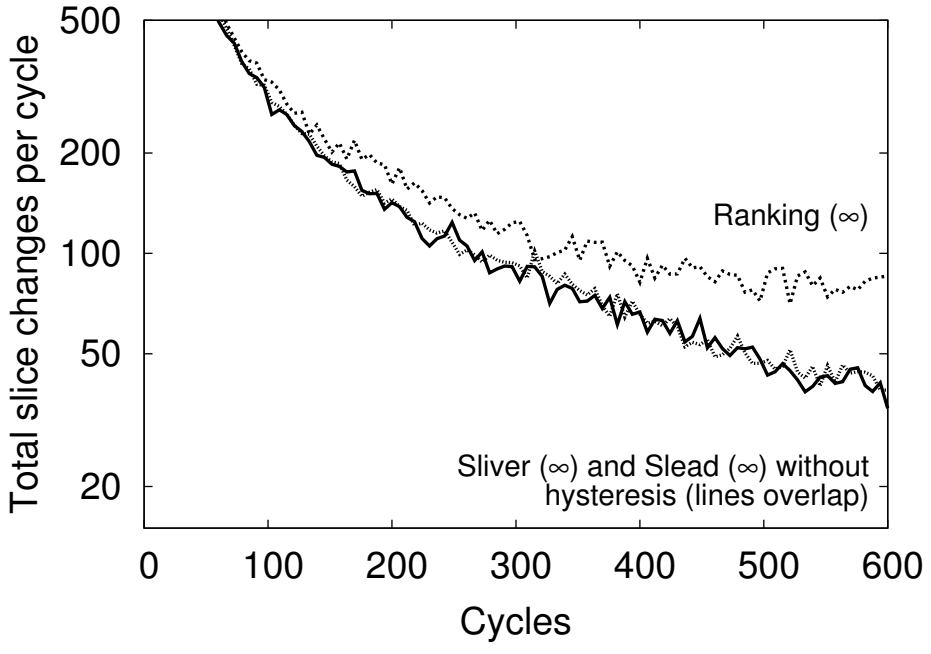


Figure 4.7: Bloom filter's impact on *steadiness*. Evolution of the number of slice changes (10,000 nodes, 10 slices).

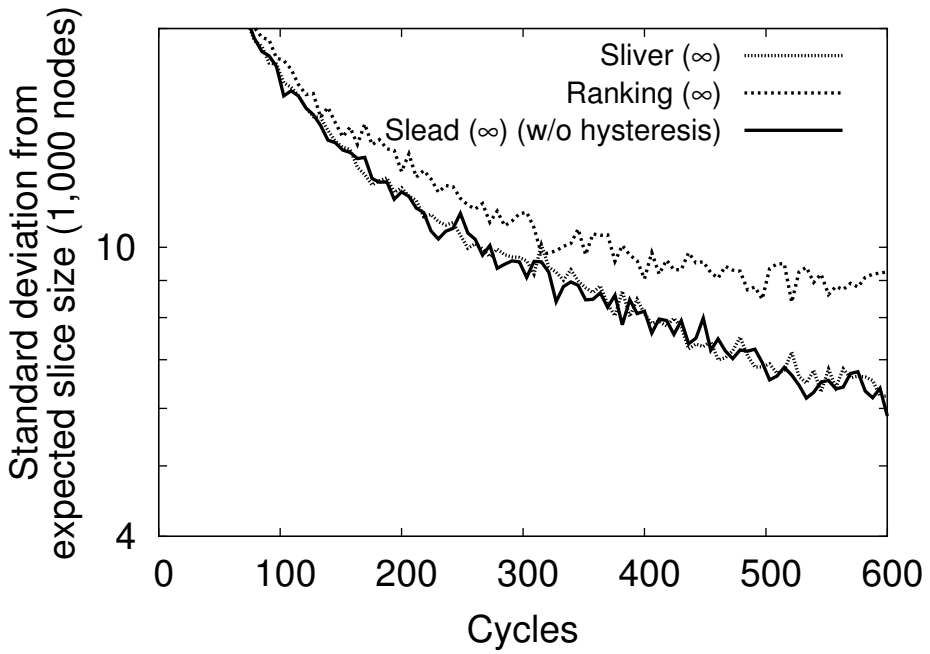


Figure 4.8: Bloom filter's impact on *slice variance*. Evolution of the slices std. dev. from 1,000 nodes (10,000 nodes, 10 slices).

4.3.3 Dynamics

In the previous section we intentionally omitted details regarding the Bloom filter implementation. Actually, such implementation impacts the behavior of the protocol, which can be tuned to meet application specific criteria.

A traditional Bloom filter implementation [Bloom, 1970] does not have the ability to delete entries. In the static scenarios we considered previously, such capacity is not required and moreover, due to the low memory consumption, this simple Bloom filter implementation copes with our requirements. However, in scenarios with churn, the ability to delete entries is required as it enables old values to be pruned enabling adaption to new configurations. In Ranking and Sliver this is addressed by the sliding window mechanism, which simultaneously limits memory usage.

In Slead we decouple these distinct but related properties simply by considering a different implementation of the underlying Bloom filter. To this end we use an implementation able to forget and mimic the sliding window-type behavior found in Ranking and Sliver. The approach used, known as A^2, provides least-recently-used semantics while keeping low memory usage [Yoon, 2010]. In short it uses two traditional Bloom filters that are

filled out of phase, i.e. one starts to be filled only after a number of updates to the other. This allows each Bloom filter to record a set of values that differ in the timeline they represent, where one contains the more recent items and is a subset of the other. The old values are deleted by judiciously swapping and flushing the Bloom filters [Yoon, 2010].

In our experiments we used the A^2 implementation with the parametrized memory size. Figure 4.9 presents the evaluation of SLEAD under a dynamic environment and thus the impact of A^2. We start with a system with 100 nodes, let it stabilize, and then at cycle 140 add 10 nodes per cycle for a duration of 10 cycles. As it is possible to observe, SLEAD exhibits similar behavior to Sliver and Ranking. Even though it incurs in slightly higher variance initially, it quickly converges and accommodates the system size changes. Moreover, when the hysteresis mechanism is added, the same quick convergence is observable validating that our complete approach is also adequate for dynamic environments.

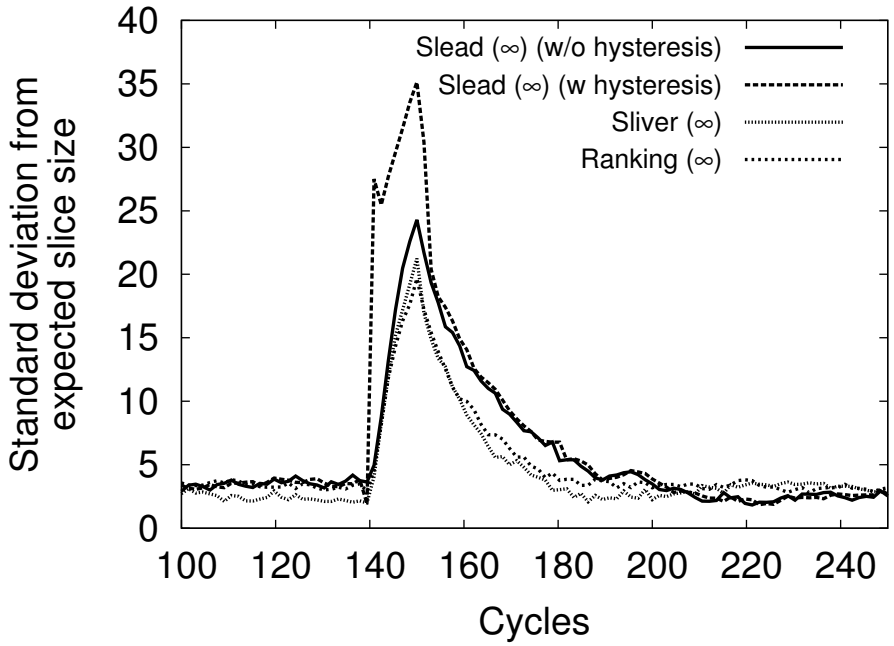


Figure 4.9: **Slice variance**. Evolution of the slices std. dev. under churn. Starts with 100 nodes, ends with 200

4.4 Slicing as a distributed systems primitive

Autonomous and fully decentralized slicing using gossip-based protocols received some attention recently [Fernández et al., 2007; Gramoli et al., 2008; Maia et al., 2012] due to its convenience and desirable properties for large-scale distributed system provisioning: dependability, scalability and adaptivity. However, little effort has been made to consider slicing as a building block for other applications and in particular, concerning its completeness. This capability is the key for composing gossip-based protocols into more complex services [Rivière et al., 2007].

We begin by extracting from the existing literature the main characteristics of slicing protocols and factoring them in a generic slicing framework. Next, we go through existing slicing protocols, instantiating our framework to compare and differentiate them. Finally, we propose a variant of the Slead protocol recurring to the modularity of the framework.

Recalling from previous sections, slicing is the process of organizing the set of nodes in a distributed system, into k groups called *slices*. Each slice must eventually be composed of the nodes that lie in the sequence of k subgroups ranked by increasing values of a sortable metric: if slices are $S_1, \ldots, S_i, \ldots, S_k$, then all nodes belonging to S_i must have a greater value for the metric than those in S_{i-1}, and a lower value for the metric than the nodes in S_{i+1}. Examples of sortable metrics include the uptime, available disk space, CPU or other application-specific metrics. The system is modeled as a set of nodes connected through an asynchronous network.

Slicing protocols operate by means of gossip-based message exchanges of partial information about the system state, that yield a global convergence but do not require any centralized knowledge. In detail, each node in the system has access to a local attribute value representing the measured value of the metric of interest (disk space, uptime, etc.). Periodically, it contacts some peers and exchanges its attribute with them. Through this mechanism, each node gathers some local knowledge that it uses to progress. This is a key characteristic of gossip-based protocols that confers them high resilience and scalability.

The set of peers each node may contact (the *view* of each node) is given by an underlying protocol called the Peer Sampling Service (PSS) [Jelasity et al., 2007]. The PSS is typically implemented using gossip-based protocols itself and produces a random stream of peers drawn from the whole system.

The PSS is a key service that maintains membership of nodes to the system in a decentralized fashion and offers a set of desirable properties, namely: i) departed nodes are eventually removed from the random stream of nodes provided at alive nodes, ii) new nodes are inserted in these streams

within a bounded amount of time, and iii) convergence is guaranteed for protocols built on top of the PSS by ensuring that all nodes will be involved in the exchanges regularly. In our experiments, we assume the availability of the Cyclon [Voulgaris et al., 2005a] PSS implementation, that provides good randomness properties for the constructed views.

4.4.1 Slicing Framework

We have defined the basic framework of a slicing protocol, with which we can instantiate the various existing slicing protocols. The pseudo-code for this framework is presented in Algorithm 4. In this algorithm, *node* represents the node *id* while v represents its attribute value.

At its core are two threads, a passive (lines 2 to 6) and an active one (lines 8 to 10), running at each node. The active thread periodically and proactively sends to each neighbor in the *view* a message containing the unique node identifier, *me*, and the current value of the local attribute, *local_attribute_value* (line 10). Recall that *view* is populated by the underlying PSS and is composed of a random subset of system peers (neighbors). The reception of those messages triggers the passive thread waiting condition. Upon reception, the slicing protocol stores the received information in a data structure (line 3) that offers three methods. The first method is $insertData(sender, attribute\_value)$ used to store incoming data. Methods $getSmaller()$ and $getTotal()$ refer to the attribute values the node has seen. The first one returns the number of attribute values which are smaller than the local one while $getTotal()$ returns the total number of attribute values received. Note that this only represents locally gathered information and does not requires global knowledge. With this local knowledge, nodes rely on an *estimate_slice*() (line 6) function to compute the node's slice and report it to the application.

What differentiates each instance of such framework is the possibility of implementing the *estimate_slice*() function and the data structure differently. Moreover, as we will see next, from the implementation details of both, the behavior and properties of each protocol change. However, in all protocols considered in this section the *estimate_slice*() function is implemented similarly (as shown in Algorithm 5). Existing literature on slicing only considers the case where every slice is equally-sized and each node in the system knows the number of slices, k, *a priori* by configuration. Consequently, computing the slice position is simply a matter of multiplying the node's position obtained from the ration of smaller attribute values and total nodes seen by k.

To the best of our knowledge, there are three main slicing algorithms in

```
1  function passive_thread
2      upon reception(message)
3          data.insert(message.sender, message.value)
4          smaller ← data.getSmaller()
5          total ← data.getTotal()
6          slice ← estimate_slice(smaller, total)
7  function active_thread
8      periodically
9          for (node, v) ∈ view do
10             send(sender = me, value = local_attribute_value)
```

Algorithm 4: Slicing Framework.

```
1  function estimate_slice(smaller, nodes_seen)
2      position ← smaller/nodes_seen
3      slice ← position * number_of_slices
4      return slice
```

Algorithm 5: Basic slice estimation.

the literature: Ranking [Fernández et al., 2007], Sliver [Gramoli et al., 2008] and Slead [Maia et al., 2012]. We describe each one of them instantiating the data structure implementation from our slicing framework and point out their specificities and motivations. Subsequently, we present a novel slicing protocol called DSlead.

Ranking. The Ranking protocol [Fernández et al., 2007; Gramoli et al., 2009] was the first slicing algorithm proposed in the literature. Its data structure simply consists on two variables: *smaller* and *total* updated each time a message is received. The pseudo-code for this data structure is presented in Algorithm 6.

It is important to note that the second part of the boolean expression in method *insertData* (line 6) is used for disambiguation. Considering the possibility of two nodes sharing the same attribute value, their *id* is used to order the nodes, improving slice calculation.

Although very simple, the Ranking protocol is highly resilient. Message loss and churn do not prevent the protocol from progressing. However, some details prevent the protocol from achieving optimal results. In particular, there is no regard to duplicate messages. A node that receives duplicate

```
1  initialization
2      smaller ← 0
3      total ← 0
4  function insertData(sender, attribute_value)
5      if ((attribute_value < local_attribute_value)
6          ∨(attribute_value = local_attribute_value
7              ∧sender < me)) then
8          smaller ← smaller + 1
9      total ← total + 1
10 function getSmaller()
11     return smaller
12 function getTotal()
13     return total
```

Algorithm 6: Data structures for RANKING.

messages from a specific peer will consider them repeatedly when calculating the slice estimative: the number of nodes with higher and lower values will thus be miscalculated leading to wrong slice attributions.

Observe that, in [Fernández et al., 2007], a RANKING node contacts a single selected node at a time. However, RANKING can be implemented by sending the attribute to all nodes in the view. Such implementation is faster and avoids biasing the protocol towards nodes in the slice border [Gramoli et al., 2009]. In our framework, we only consider this version of the protocol.

Sliver. To solve the duplicate message problem, the SLIVER [Gramoli et al., 2008] protocol was proposed. It is very similar to RANKING but alongside the node attributes, SLIVER also stores the node identifiers. This way, attributes from a specific node are considered only once in the slice estimation. The pseudo-code for SLIVER's structure follows on Algorithm 7. The data structure to support SLIVER is a key-value table where the keys are node ids and values their attributes (line 2).

It is possible to see that this protocol converges under the assumption of the availability of a PSS. Let us consider that a single node is capable of storing a number of pairs $(id, attribute\_value)$ equal to the size of the system. Due to the random nature and continuous refresh of the PSS views, eventually every node will receive a message from every other node in the system. With this global information available at each node it is easy to see that the protocol will converge and every node will compute the correct slice.

```
1  initialization
2      list ← new dict()
3  function insertData(sender, attribute_value)
4      list[sender] ← attribute_value
5  function getSmaller()
6      res ← 0
7      for k, v ∈ list do
8          if (v < local_attribute_value)
9              ∨(v == local_attribute_value
10                 ∧ k < me) then
11             res ← res + 1
12     return res
13 function getTotal()
14     return list.size()
```

Algorithm 7: Data structure for SLIVER.

At this point it is important to make an observation. As noted in Section 4.1, a protocol that relies on an amount of information proportional to the system size is not scalable nor suitable to large-scale systems. To address this problem and to make SLIVER run in a bounded memory environment, instead of storing all received attribute values, only the more recent ones are kept. In practice, this is achieved by considering a FIFO queue with fixed size. It should be noted however, that this adjustment not only solves memory issues but also allows the protocol to handle churn in a more effective way. Nodes leaving the system will stop publishing their attribute values and the limited size queue will force them to be eventually forgotten from the system. Analogous behavior happens for nodes joining the system.

An alternative implementation of the RANKING protocol can also be considered in order to allow the protocol to forget attribute values. Instead of simply storing two variables, a fixed size list of attribute values is stored. The core behavior of the protocol is preserved but now it is able to cope with dynamic attribute values.

Slead. The SLEAD [Maia et al., 2012] protocol was proposed with the objective of tackling the lack of steadiness and high memory consumption issues manifested by previous approaches.

Steadiness issues were addressed using an hysteresis mechanism. In prac-

tice, the mechanism delays slice change decisions until such decisions have been confirmed by more than one cycle of slice estimation. This mechanism avoids unnecessary slice changes, specially for nodes at slice borders. The hysteresis mechanism, detailed in [Maia et al., 2012], is pluggable to all slicing protocols and is left out of the scope of the present version of the framework.

Memory consumption problems arise from the need to store a list with every pair of $(id, attribute\_value)$ received in order to ensure that the protocol converges. Let us consider a stable environment where each node has a constant attribute value and no node leaves or enters the system. It is easy to see that, in such scenario, in order for a slicing protocol to converge to the correct slice organization it is necessary that each node *sees* the attribute value of every other node in the system. With that complete view over the system it is possible to compute the exact slice to which the node belongs. Nevertheless, having a protocol that uses an amount of memory proportional to system size is clearly not scalable. This problem was addressed with the use of a FIFO queue of fixed size, m, that follows the behavior of a sliding-window. As a result, at each point in time, every node has access to a sample of m pairs $(id, attribute\_value)$ with which it is able to estimate its relative position and thus its slice. Because the underlying Peer Sampling Service provides a stream of nodes that is extremely close to a continuous random selection, it is expected that each network sample preserves characteristics similar to the system as a whole distribution, hence allowing each node to correctly estimate its slice. However, in practice the size of m impacts the accuracy and steadiness of the protocol. Low values of m degrade the quality of the sample and negatively impact the protocols behavior while high values of m result in high memory consumption rates. This behavior is observable in the results from [Maia et al., 2012].

SLEAD's solution to the high memory consumption was the use of Bloom filters [Bloom, 1970] to store data. With Bloom filters, SLEAD is able to store the complete view of the system with a bounded memory footprint which solves the problem for the case of a stable environment. Still, as noted in the SLEAD paper, the sliding-window-type behavior of SLIVER not only addresses memory consumption issues but also addresses system dynamism. The system may experience instability due to two main factors: churn or node-level change of attribute values. Both these factors provoke the need for the system to adapt and consider new information arriving and *forget* obsolete one. In SLIVER, this is immediately achieved through the fixed sized queue as old values are progressively being forgotten and replaced by fresh data. In SLEAD, as traditional Bloom filters do not have the capacity to remove items, a special kind of Bloom filters, called A^2 [Yoon, 2010] is used. This Bloom filter implementation is capable of forgetting values by having

```
1  initialization
2      smaller ← new A2BloomFilter()
3      greater ← new A2BloomFilter()
4  function insertData(sender, attribute_value)
5      if ((attribute_value < local_attribute_value)
6          ∨(attribute_value = local_attribute_value
7              ∧sender < me)) then
8          smaller.insert(sender)
9          greater.remove(sender)
10     else
11         smaller.remove(sender)
12         greater.insert(sender)
13 function getSmaller()
14     return smaller.size()
15 function getTotal()
16     return smaller.size() + greater.size()
```

Algorithm 8: Data structures for SLEAD.

two Bloom filters and periodically reseting one of them. Beside, we replaced the traditional Bloom filters used in A^2 by counting Bloom filters that allow for the removal of specific items [Fan et al., 2000]. The combination of both mechanisms enables SLEAD to cope with dynamism.

The instantiation of SLEAD in our slicing framework is presented in Algorithm 8.

DSlead: Decaying Slead. Although successful in reducing steadiness and memory consumption issues from previous approaches, the SLEAD protocol exhibits some frailties. In particular, the way it deals with dynamism with the A^2 Bloom filter implementation has a main disadvantage. related to the difficulty of configuring the rate at which values are being forgotten in a tractable way. This is important since forgetting values too fast results in protocol output instability while forgetting them slowly results in very slow adaptation to change. In SLIVER, changing this rate is easy because it suffices to change the queue size (a larger queue means slower adaptation to change). In SLEAD this is not a straightforward task. The rate at which the protocol forgets values is related to the fill ratio of the A^2 Bloom filter [Yoon, 2010]. Inevitably, this means that changing the refresh rate is achieved by

changing the Bloom filter size. This is definitely not practical as changing the Bloom filter size means reseting the whole Bloom filter, impacting negatively on the slice estimation.

Fortunately, we can take advantage of the slicing framework and SLEAD modularity. In fact, SLEAD is independent of the Bloom filter implementation. To solve these issues and achieve a more complete slicing protocol we propose the use of a different Bloom filter variant: time-decaying Bloom filters [Cheng et al., 2005]. These Bloom filters not only allow direct item removal but also allow the user to define a function that removes according to a certain time-related function. In other words, it is possible to define at which rate items are forgotten from the Bloom filter. This leads to DSLEAD, a variant of SLEAD that shares the same structure from Algorithm 8 but with a different Bloom filter implementation.

In our implementation it works as follows. Each time an item is inserted into the Bloom filter a certain number of positions in the Bloom filter array are incremented according to a set of hash functions [Bloom, 1970]. It is important to note that the implementation of time-decay Bloom filters relies on counting Bloom filters [Fan et al., 2000] which have more than one bit per array position, allowing to count various occurrences of the same item and enabling item removals. Then, periodically, each of the array positions is multiplied by a $fraction$ value ($[0, 1[$). If a certain position or group of positions in the array are not refreshed their value will eventually decay to a value close to 0. As the value never reaches 0 and, in order to actually *forget* items, when a certain value in a certain array position becomes smaller than a user defined threshold, it is considered to be 0.

A decay function is, therefore, composed by three variables: the period of decay, the fraction of decay per period and the minimum threshold. The ability to define a decay function over the Bloom filter values and easily change the rate at which it is operating, alongside the ability to immediately accommodate changes to attribute values completes the DSLEAD protocol.

We evaluate both SLEAD and DSLEAD in the following experiment. We measure protocol steadiness as the number of slice changes that occur in the system for a particular cycle [Maia et al., 2012]. In our experiment we let the protocols run for about 50 cycles and then triggered a configuration change increasing the memory footprint of SLEAD and decreasing the decay rate of DSLEAD. The results are depicted in Figure 4.10. Note that we intentionally configured SLEAD and DSLEAD with small memory and high decay rate respectively which issue a non desirable behavior from both protocols, observable until cycle 48. In particular, the steadiness of slice estimation is degraded as values are continuously being *forgotten* or *decayed*. Both protocols eventually converge, lowering the steadiness values. However, DSLEAD

does not incur the same burst of slice changes as SLEAD. This is due to the fact that in order to reconfigure SLEAD it is necessary to reset the protocol's data structure while this is avoided in DSLEAD, resulting in a much smoother transition.

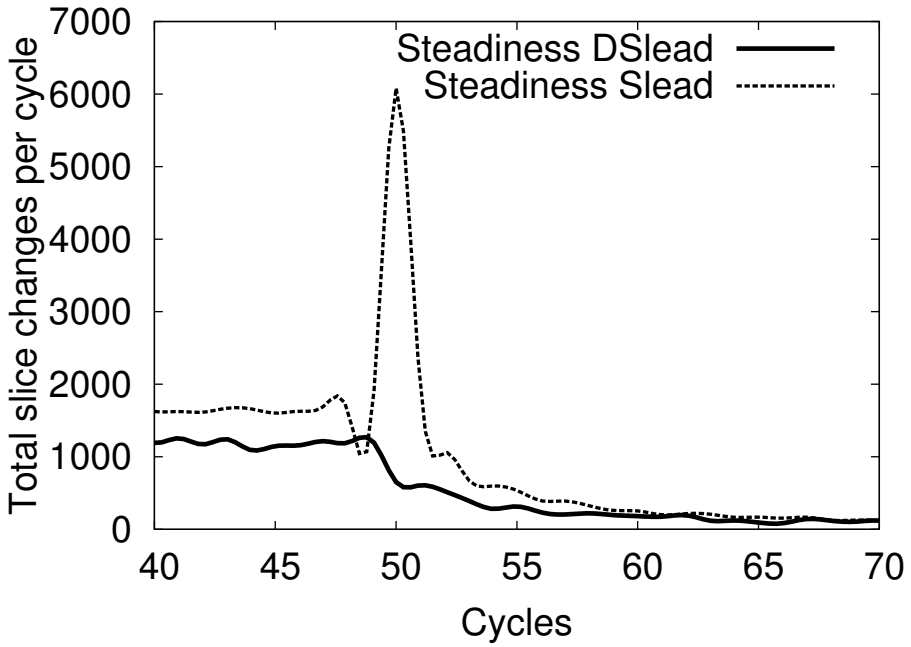


Figure 4.10: SLEAD and DSLEAD with reconfiguration.

4.4.2 Extending Slicing

Slicing protocols provide the capability of assigning each node a slice which is useful from the perspective of system organization. However, nothing is said about how nodes from the same slice interact with each other or how to have different sized slices. These are very important features for slicing protocols to be practical.

In particular, in our opinion, there are four main features that are lacking from slicing protocols. First, it is important to support slices with different sizes. This feature widens the range of applicability of slicing protocols. It considers, for instance, the case of having 10% of nodes with a coordination task and groups of 30% nodes each with separate responsibilities. Secondly, having a fixed slice configuration also impairs the applicability of slicing protocols. These protocols are intended for highly dynamic environments where adaptivity is key. Consequently, it is difficult to conceive the use of a protocol

that needs to be pre-configured and restarted each time a new configuration is needed. We propose the addition of a mechanism that allows slicing protocols to change slice configuration without having to stop or restart the system. Third, in [Jelasity and Kermarrec, 2006] it is noted that slicing is only useful if each slice is presented to the application as a group. This means that each node in the system should not only know to which slice it belongs to but also which nodes share such slice. Finally, the ability to slice considering more than one attribute seems a very useful feature. For instance, allowing to look not only to CPU load as an indication of node utilization but to a combination of different metrics like disk I/O and uptime. In this section we describe each one of these features and how they can be implemented.

In addition, we evaluate some of these features. In this regard, we ran our experiments on top of Splay [Leonini et al., 2009]. Splay is a platform that enables rapid development and testing of distributed systems. In particular, we ran each experiment on a local deployment of Splay and each consisting in 1000 Splay nodes. Splay was deployed in a 24-core machine with 128GB of RAM. Each node runs the same Lua [Ierusalimschy et al., 2007] code consisting of the DSLEAD protocol. As described earlier, DSLEAD follows a gossip-like message exchange pattern. Each node periodically contacts its neighbors sharing its locally read attribute and considers this a *cycle*. However, nodes are not synchronized which means there is no guarantee that nodes are progressing, in terms of cycle count, at the same rate. In order to retrieve usable information from system runs these logical cycles are ignored and used only internally by each node. We present our results in terms of cycles measured in actual time. This period of observation was configured to be of 10 seconds and a cycle, in our experiments, should be understood as one of these periods.

Heterogeneous slicing. Previous work on distributed systems slicing protocols focused in dividing the system into k equally-sized slices. However, this approach is restrictive. Moreover, it is possible to equip existing protocols with the capability of considering different slice configurations with minimum change to the protocols and maintaining their properties. We name these slice configurations *schemas*.

Originally, each node calculates its slice by estimating its position in a virtual ranking of all nodes according to a certain attribute. This position estimative is calculated dividing the number of smaller attributes observed by the total number of attributes observed as described in the *estimate_slice*() function implementation of Algorithm 5.

To allow the protocol to consider different slice schemas, we need to store

```
1 initialization
2   CL ← list with slice configuration
3 function estimate_slice(smaller, total)
4   position ← smaller/total
5   slice ← 0
6   for s in CL do
7     if position <= s then
8       return slice
9     slice ← slice + 1
```

Algorithm 9: Implementation of heterogeneous slice estimation.

an additional data structure representing the slice size distribution: a simple schema configuration list (CL) with cumulative percentages is sufficient. This list is populated with one entry per slice (s_1 to s_k) and each entry, i, represents the percentage of the system expected to be assigned to all slices from s_1 to s_i. For instance, the list $CL \leftarrow [0.2, 0.4, 1]$, issues a system partitioned into three slices. The first slice would gather 20% of the system, the second slice another 20% of the system and the third slice would encompass the remaining 60% of system nodes. It is important to note that this organization still follows the virtual ranking of nodes according to a local attribute.

Furthermore, even though we are using DSLead in our experiments, this technique can be implemented by all the slicing protocols that fit the slicing framework we defined in Section 4.4.1 simply by reimplementing *estimate_position* as follows (Algorithm 9). The new function will still take as arguments the total number of attributes seen by the node and the number of those that are smaller than its local attribute. It computes the node's relative position in the virtual rank of all nodes as before but uses this result in a different way. The node slice is estimated by checking in which schema configuration interval such position falls.

This change in slicing protocols proved to be effective. To evaluate this particular feature we ran two different tests with different slice schemas. Schema one considers five slices, each with 20% of the system nodes and it was chosen in order to show that previous equally sized slices are still achievable in this new protocol version. On the other hand, schema two is an heterogeneous slice schema. It is configured to achieve three slices, one with 50% of the nodes and the remaining two with 25% of nodes each. The results are depicted in Figure 4.11 and Figure 4.12, respectively.

Each vertical bar represents how the slices are distributed in a certain

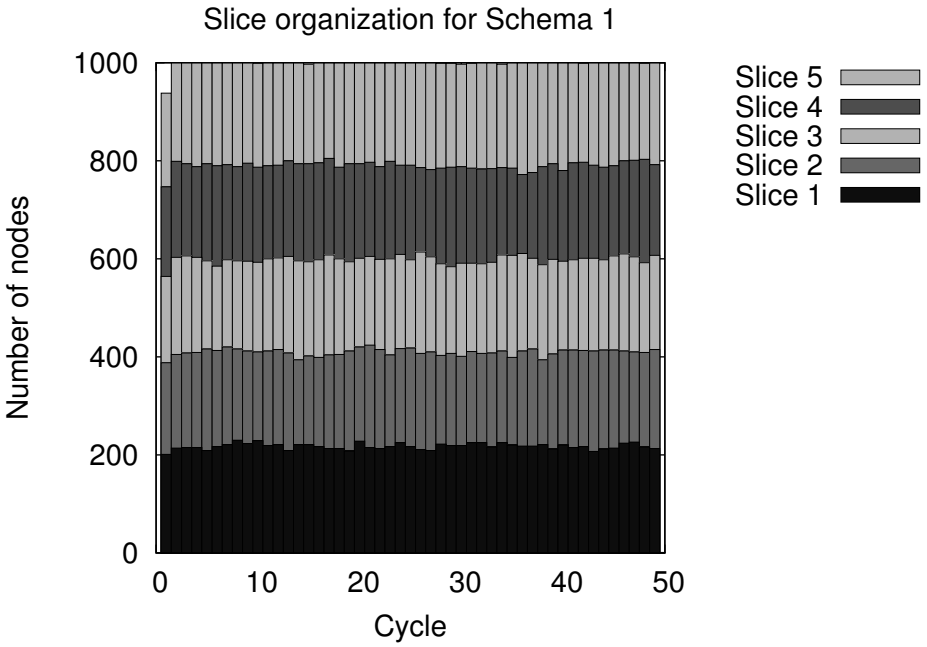


Figure 4.11: DSLEAD run configured with schema one.

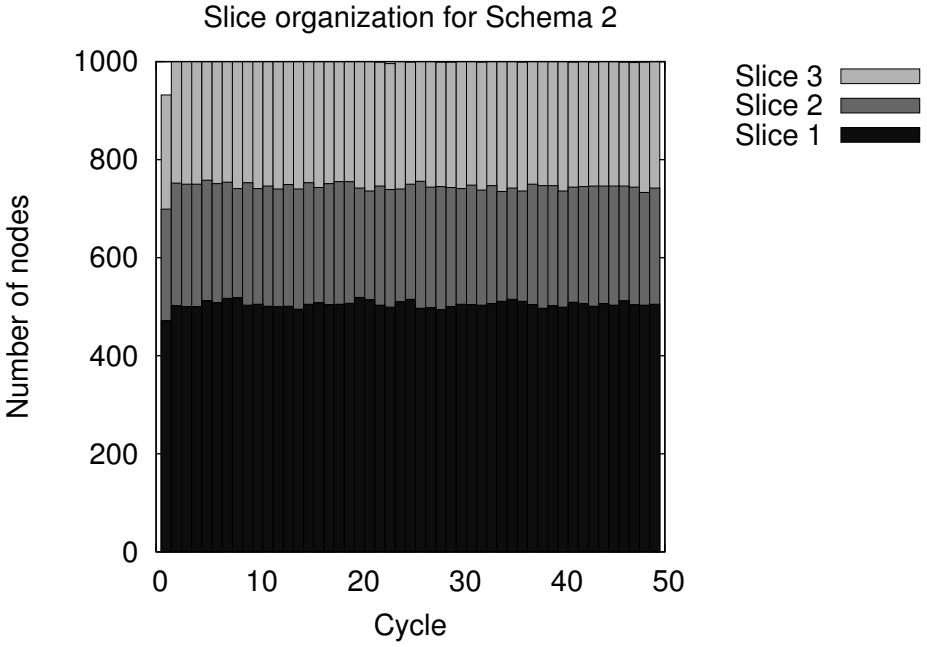


Figure 4.12: DSLEAD run configured with schema two.

```
1 function changeSchema(newSchema)
2     if not CL == newSchema then
3         CL ← newSchema
4         for peerinview do
5             send(peer, newSchema)
```

Algorithm 10: Implementation of *changeSchema* function.

cycle. Each shade of grey represents the amount of nodes of a certain slice. Slices as ordered from bottom up according to the CL list order. The sum of all segments of the vertical bar is the total number of nodes in the system. Note that this is not true for some initial cycles, as can be observed in Figure 4.11 and Figure 4.12 where some nodes are still starting.

Slice reconfiguration. In previous work [Fernández et al., 2007; Gramoli et al., 2008; Jelasity and Kermarrec, 2006; Maia et al., 2012] focused on slicing, the number of slices was a pre-configured parameter and little is said about dynamic reconfiguration of slices. Moreover, now that it is possible to have different slice schemas, the logical step is to capacitate slicing protocols with the ability to change schema *on-the-fly*.

We begin by assuming that the initiative of changing schema is external to the system and communicated to an arbitrary node or set of nodes. The nodes that receive the schema change request are responsible for processing it. The main challenge here is how to effectively disseminate the schema to all system nodes. However, all the slicing protocols rely on Cyclon [Voulgaris et al., 2005a]. Cyclon provides each node a random view over the complete set of nodes and this view has the properties of a connected graph. As a result, in order to send a message to all nodes in the system, it is sufficient to send it to all the nodes in the current Cyclon view and have each node repeat such task on the reception of a new message, essentially flooding the network. It is important to note that more elaborate and effective dissemination techniques could be used to spread the slice reconfiguration message [Carvalho et al., 2007]. Nonetheless, such messages are very small in size and sent sparingly and thus we consider such optimizations out of the scope of this work. The code for schema change request is presented in Algorithm 10.

In order to validate this approach we ran DSlead with an initial slice schema with $CL \leftarrow [0.1, 0.5, 0.6, 0.7, 0.9, 1]$ and issued a schema change to $CL \leftarrow [0.1, 0.2, 0.3, 0.4, 0.5, 1]$ at cycle 250. The change request was made to a single node in the system, responsible for propagating it.

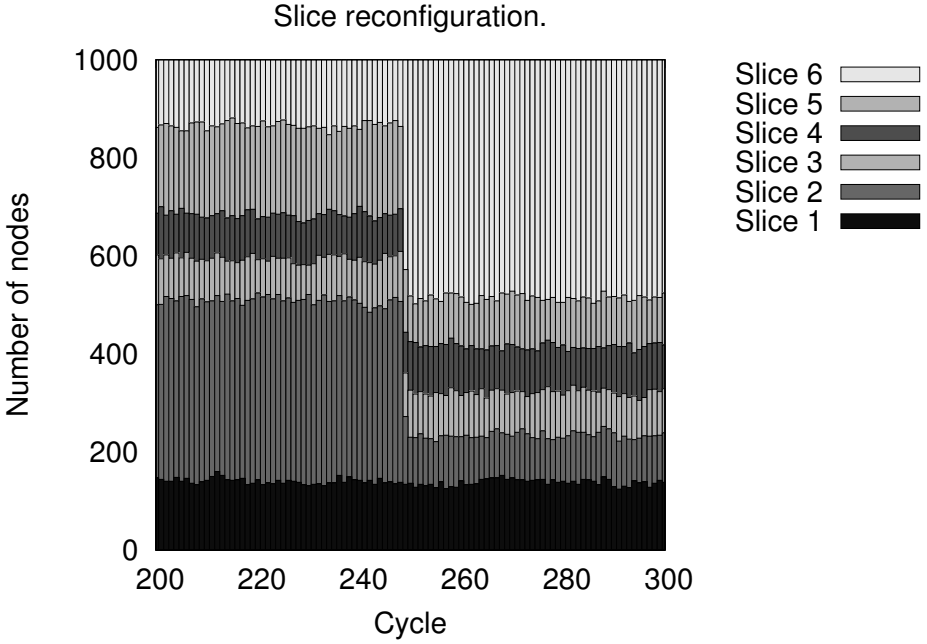


Figure 4.13: Slice reconfiguration.

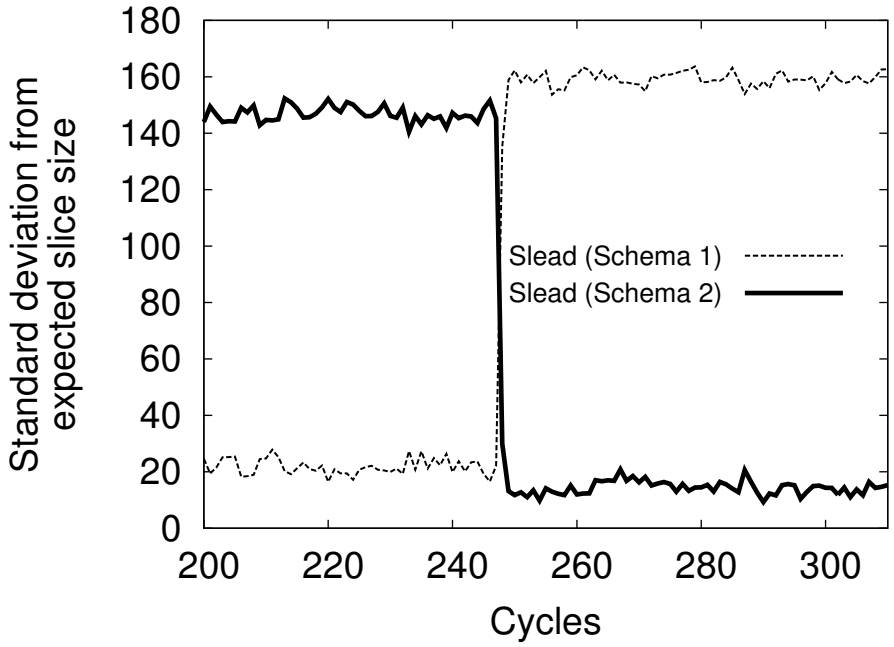


Figure 4.14: Slice variance for a slice schema change.

To better illustrate the protocol behavior, Figure 4.13 depicts slice distribution at each ten cycle period and Figure 4.14 depicts slice variance measurements. Slice variance, as defined in [Maia et al., 2012], measures the distance of a certain slice distribution to a target distribution. In our experiment we determined the system slice variance according to both schemas. It is interesting to see how the two curves behave when the schema change occurs. Initially, as expected, high variance values are measured for schema two and low variance values measured for schema one. Around cycle 250, the curves exchange roles indicating the schema change. The same time behaviour can be observed in Figure 4.13. For instance, if we look at slice two, which held 40% of the nodes initially we can see how it shrinks to the 10% of nodes configured in schema two.

Slice-local view. Independent of the slice schema discussion, another gap in current slicing protocols is intra-slice connectivity. In the current slicing protocols, each node is capable of communicating with a set of other nodes in its *view* which are called neighbors. This group of neighbors, although of key importance, is oblivious to the slicing protocol. Consequently, there is no practical way for a node to contact its slice peers. In fact, partitioning a distributed system into slices is only useful if it is possible to take advantage of such slices.

In order to achieve this the immediate solution is to have another instance of a Peer Sampling Service for each slice. This approach was actually mentioned in [Jelasity and Kermarrec, 2006] as a future research path. This has however a bootstrapping problem as we do not know in advance each node's slice. Our approach is to inject the node's current slice into the slicing protocol message defined in Algorithm 4, line 10. Besides the current node's slice we also inject the slice-local view when the target node belongs to the same slice as the sender. This information, despite limited, allows nodes to quickly populate its slice-local view and quickly deal with changes by resetting it when slice changes happen.

Multi-attribute slicing. Existing slicing strategies take into account a single system attribute to rank nodes. However, it might be useful to consider more than one attribute in order to better characterize each node. For instance, considering various load attributes simultaneously or considering attributes such as geographic information and bandwidth simultaneously. The immediate approach can be to extend the slicing protocol to exchange more than one attribute at each algorithm cycle. This would allow considering various attributes for slicing. We propose instead a local

computation of an artificial attribute resulting from the combination of different measured attributes and in particular the use of Space Filling Curves (SFC) [Sagan, 1994]. This mathematical construction provides a mapping from a d-dimensional space to a unidimensional one. The different attributes considered are viewed as a multi-dimensional space. This space is divided into sub-spaces which are mapped to a line that traverses the sub-spaces passing through every point and entering and exiting the space only once. Then, the virtual attribute can be constructed as the length of the line from one of its ends to the point with spatial coordinates derived from the attribute values. Therefore, nodes will be able to map several attributes to a single point in the SFC given by a real number. This number is then used as the ranking criteria, allowing existing protocols to run unmodified but still supporting multi-attribute slicing. An example of the application of these curves can be found in [Vilaça et al., 2011].

Moreover, in the follow up paper [Pasquet et al., 2014], we study the Space Filling Curve approach for multi-attribute slicing and propose an additional solution for the problem based on the notion of dominant relations. This work was done out of the scope of the present thesis.

4.5 Discussion

In this chapter we focused on distributed systems slicing protocols. Slicing protocols autonomously organize an arbitrary number of nodes into groups according to a given criteria. Moreover, this process is completely decentralized and can scale to systems with millions of nodes. These characteristics made slicing our first choice as the implementation of the group construction component of DataFlasks. However, some frailties in state-of-the-art slicing protocols impeded their immediate application in such context. Among these frailties were problems of instability, memory consumption and the inability to consider slices (groups of nodes) with different sizes. Consequently, we proposed Slead and DSlead to solve the various issues identified in previous protocols. Additionally, we propose a slicing framework that can be instantiated in order to take the form of every slicing protocol. This allows for reasoning about the protocols, clearly identify the differences between them and identify the key constructions needed to design new ones.

The most important requirement for the group construction component of DataFlasks is the ability to divide nodes into groups in a scalable and robust way. Moreover, each node must know the group it belongs to. Distributed systems slicing meets these requirements. However, group size is determined as a given percentage of the system. As described in Section 4.4.2,

slicing protocols receive as an input a slice schema that defines the number of slices to consider and the percentage of nodes assigned to each slice. Slice size is thus dependent on system size which is considered to be unknown. Once that, in DataFlasks, slicing should be used for data distribution and replication this represents a frailty of the approach. In fact, the replication factor is not directly controllable by the system administrator. This impairs the ability of the administrator to intervene in the system and to adjust the replication factor according to the characteristics of the production environment. Considering this limitation, we worked on a new protocol for group construction that would allow the replication factor to be configurable. Such work is presented in the next chapter.

Even though the work on slicing stems from its applicability for the group construction component of DataFlasks, this chapter is a contribution in itself as our protocols are completely independent from the design of DataFlasks. Moreover, the slicing framework and the new features introduced with DSlead are usable in other contexts and are intended as a building block for other applications and services.

Chapter 5

Group Construction Protocol

A fundamental component of DataFlasks is group construction. DataFlasks nodes must be divided into groups in order to distribute data and every node must learn to which group it belongs. In this chapter, we present an algorithm that is able to organize thousands of nodes into groups in a robust and scalable way. The algorithm receives as an input the size of the groups to construct. This size is user defined and, by configuration, every node in the network learns the same desired group size at start up.

We begin with some remarks that give the intuition behind the design of the algorithm. Next, we present a simplified version of the algorithm and sketch a proof of its correctness. In Section 5.2 we show how it can be extended in order to be faster and more effective. Section 5.3 concludes the chapter.

5.1 The basic protocol

In the design of the protocol it is important to take into account that its main goal is to divide nodes into groups for data distribution and replication. In this scenario, each time a node changes group it needs to perform state transfer procedures. The design of our group construction algorithm aims at minimizing these procedures, which are costly. To this end, we designed the algorithm to always consider the number of groups to be a power of two. Consider Figure 5.1. Forcing the number of groups ($ngroups$) to be a power of two, results in a well defined set of possible group configurations. Each configuration is associated with a level number where $ngroups = 2^{level}$.

An important thing to notice is that the mapping between the key and group is stable as the level increases. Recalling Section 3.3, data distribution is done by assigning key-ranges to each group by the use of an hash

```
1 Method group(key):
2     key_hash ← hash(key)
3     key_position ← key_hash/hash_max_value
4     group ← ⌈key_position ∗ ngroups⌉
5     return group
```

Algorithm 11: Determining to which group a certain key-value pair belongs.

function (*hash* in line 2 of Algorithm 11). The hash function maps the keys with arbitrary range size into keys of fixed range size, trying to do so as evenly as possible over the target range. Assuming the target range is]0,hash_max_value], it is possible to map each key to a position in the range]0,1] (lines 2 and 3 of Algorithm 11). With this mapping, it is straightforward to calculate the group a key belongs to (line 4 of Algorithm 11). Once each key is mapped to a]0,1] range, its position is preserved across different configuration levels as depicted by the black arrow in Figure 5.1. The goal of this design is to minimize state transfers between nodes every time there is an group change. In fact, when a configuration level is increased, nodes do not need to transfer any data. Deleting spurious data is even optional and may be performed only if space is needed. Conversely, when a level is decreased, state transfers are made only between pairs of groups distributing and balancing the task.

Algorithm. Typically, a gossip protocol works as follows. Every node knows a set of other nodes in the network, which we call *view*. Periodically, each node contacts one ore more nodes in its view and shares knowledge with them. Through these periodical exchanges each node is able to gather sufficient information to progress. Strikingly, many gossip protocols are effective even if the size of the view only grows logarithmically with the size of the system. This characteristic renders these protocols highly scalable. Even so, the node view must be populated. This problem is addressed by a specific class of protocols, which are themselves gossip protocols and which implement a Peer Sampling Service [Jelasity et al., 2004, 2007; Voulgaris et al., 2005a,b]. These protocols provide each node with a random stream of peers which is used to populate the node view. Our group construction protocol assumes the existence of such a service. In particular, we consider Cyclon [Voulgaris et al., 2005a] as the Peer Sampling Service.

Cyclon works by periodically exchanging messages containing a set of random node references from the network. These references contain the information needed to contact the corresponding nodes. For the purpose of

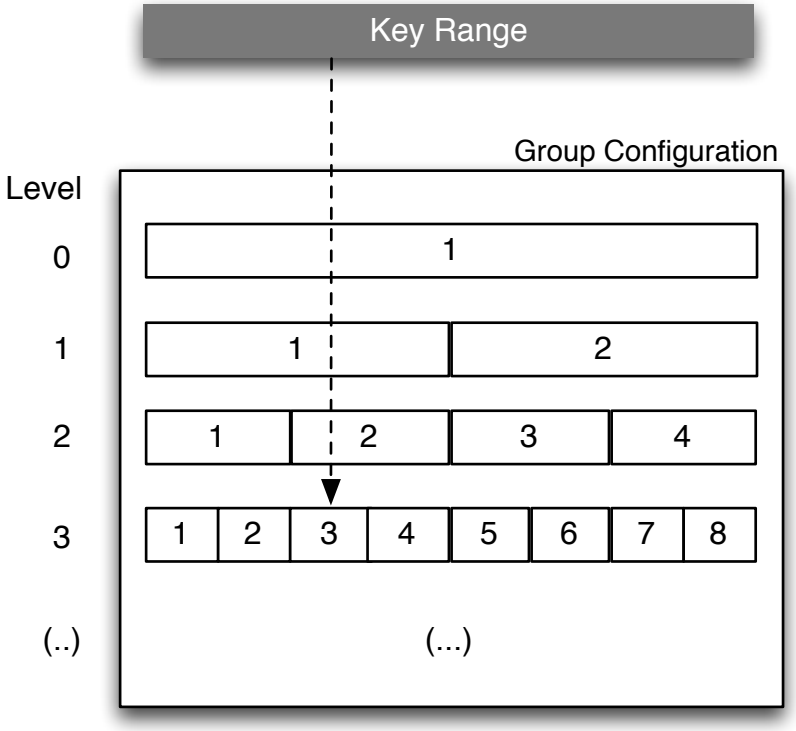


Figure 5.1: Data to group mapping and group levels.

simplicity, we consider that these references are the node identification numbers and that knowing a node identification number (id) is sufficient to be able to contact it. In our protocol we leverage the existence of the PSS taking advantage of the messages it exchanges. Each time a PSS message containing a random set of peers from the network is received, a copy of this message is delivered to our algorithm. Our protocol reacts to the reception of such messages and, solely based on them, converges to the desired groups configuration. We consider that nodes are completely connected through lossy communication channels [Guerraoui et al., 1998].

The simplified version of the group construction algorithm is presented in Algorithm 12.

The protocol has two parameters. The desired group size and the current node identification. Every node in the system runs the same protocol and is initialized with the same group size ($groupsize$). The node identification (id) uniquely identifies each node.

Upon initialization every node considers the system as a single group and that it belongs to that group. To this end it initializes variables *ngroups* and *group* with the value 1. The former variable stores the number of groups

```
   input    : groupsize, id
   Data: float pos ← random()
   /* random number in ]0, 1]                                          */
   Data: ngroups ← 1
   Data: group ← 1
   Data: set localview ← {(id, pos)}
1  upon reception of m ← set of (id, pos) from PSS:
       /* add new peers to localview                                   */
2      foreach peer in m do
3          if group(peer.pos, ngroups) == group then
4              localview = localview ∪ {peer}/* possibly rewriting peer */
       /* clean localview                                              */
5      foreach peer in localview do
6          if group(peer.pos, ngroups)! = group then
7              localview = localview \ {peer}
       /* need to merge or split?                                      */
8      if |localview| < groupsize then
           /* Should Merge.                                            */
9          if ngroups > 1 then
10             ngroups ← ngroups/2
11     if |localview| > groupsize then
           /* Should Split.                                            */
12         ngroups ← ngroups ∗ 2
       /* recalculate my group                                         */
13     group ← group(pos, ngroups)
```

Algorithm 12: Gossip group construction algorithm.

the node estimates should exist to comply with the desired group size. The latter stores the estimation of the group the node belongs to. Additionally, each node has a list variable (*localview*), where it stores peers that belong in his group. Initially, the node only considers himself in this list. As the protocol runs, the estimation of *ngroups* converges towards a number that divides the system into groups of *groupsize* nodes.

An important initialization step is generating the node position (*pos*). The node position is a number in the interval $]0, 1]$ generated at node start and that remains constant while the node is alive. This value allows nodes to distribute themselves into groups. The position is calculated using a number generator, which we assume is uniformly random across the entire network. Note that, with the node position and knowing the number of groups (*ngroups*) it is trivial to calculate the group to which the node belongs. The

```
1 Method group(position, ngroups):
2     group ← ⌈position ∗ ngroups⌉
3     return group
```

Algorithm 13: Group calculation method.

node position places the node in a range]0,1]. Consequently, to calculate the node group it suffices to divide such range into *ngroups* smaller ranges and determine in which of those the node position fits. Moreover, by the uniformity of the number generator, nodes will be evenly distributed across groups. The node group calculation is abstracted in line 13 of Algorithm 12 and shown in Algorithm 13.

DataFlasks group construction algorithm works as a passive thread that waits for messages from the Peer Sampling Service, which contain references for other nodes in the network. In DataFlasks, node references also include the node position. Recall that the node position is calculated only once and remains unchanged while the node is alive. It is thus safe to disseminate the position alongside the node *id*.

Upon the reception of a PSS message (line 1), the protocol performs four tasks. First, for each node reference in the message it checks if the correspondent node belongs to the same group (lines 2 to 4). If it does, it adds such reference to its local view (*localview*). Second, it checks if every reference in its local view still belongs to the same group (lines 5 to 7). This is necessary because nodes may change their estimation for *ngroups*. Consider a scenario where a node estimates that the correct value for *ngroups* is 2. In that case, half of the system nodes belong to the same group as the node. However, if the node refines its estimation to a value of *ngroups* of 4, then only a quarter of the system nodes can now belong to its local view. Following this process, *localview* holds references for peers that each node estimates to belong in its group. Consequently, the size of *localview* is the group size estimation at each node. At the third step of the algorithm (lines 8 to 12) such group size estimation is compared with the *groupsize* defined by the user. If the current size of *localview* is smaller or greater than *groupsize* the node refines its estimation of *ngroups* in order to correct such violation. For the case it is greater that desired, *ngroups* is multiplied by two in order to lower the group size. We name this operation a *split*. Inversely, nodes perform a *merge* operation when there are insufficient nodes in the group. Finally, after adjusting *ngroups*, each node recalculates the group it belongs to (line 13).

With the continuous arrival of PSS messages the protocol continuously improves the estimation for *ngroups*. In the remainder of the Section, we

present a proof of correctness for the simplified version of the protocol and present simulation results that show it converges to the correct group configuration.

Proof of correctness. The objective of Algorithm 12 is to group an arbitrary large number of nodes into sets of size *groupsize* (being *groupsize* the desired replication factor). In the following we sketch the proof that given a stable membership then the algorithm eventually converges and stabilizes.

Let us assume N nodes, such that $\frac{N}{groupsize} = 2^{level}$ for some $level \geq 0$. These nodes do not fail or leave the system. Nodes are fully connected by lossy communication channels, have access to a Peer Sampling Service that provides each node with a periodical random sample of nodes from the entire system, and also to a uniform random number generator in the interval]0,1].

Each node manages a variable *ngroups*. We show that, starting with $ngroups = 1$, each node i will eventually reach $ngroups = \frac{N}{groupsize}$ and stabilize there. We do so by firstly 1) showing that the algorithm has an upper bound $\frac{N}{groupsize}$ on the number of groups it can split the system into, then that, 2) at each node, *ngroups* cannot be indefinitely smaller than $\frac{N}{groupsize}$, and finally that 3) eventually, once $ngroups = \frac{N}{groupsize}$, *ngroups* no longer changes.

In the following, consider that for each level l, j is a neighbor of i if it belongs to the same group of i at l. From the group calculation $group \leftarrow \lceil position * ngroups \rceil$ (line 2 of Algorithm 13) if a node j is a neighbor of i at level l then it is a neighbor of i for every level k where $k < l$.

1) The algorithm has an upper bound $\frac{N}{groupsize}$ on the number of groups it can split the system into. Assume not, that is, eventually $ngroups > \frac{N}{groupsize}$.

Let $2^g = \frac{N}{groupsize}$. Once $ngroups > \frac{N}{groupsize}$ then the node is at least at level $g + 1$. It means that the node has performed a split at level g, which means that $|localview| > groupsize$ at level g. However, this is not possible since for 2^g groups with N nodes there are at most *groupsize* nodes per group. A contradiction.

2) At each node, *ngroups* cannot be indefinitely smaller than $\frac{N}{groupsize}$. Again, for a contradiction, assume that $ngroups < \frac{N}{groupsize}$ is always true.

As $ngroups < \frac{N}{groupsize}$ then i must be at some level $k < g$. Because any neighbor of i at level k is also a neighbor of i at any level $j < k$, then by the PSS properties all neighbors of i at level k will be eventually addded to i's *localview*. These nodes will not be removed from i's *localview* (lines 5 to 7) while i is at any level $j \leq k$. Since the neighbors of i at any level $k < g$ is larger that *groupsize*, i's *localview* at level k will eventually grow

larger than *groupsize* and i splits. At level $g-1$ i eventually splits and $ngroups = \frac{N}{groupsize}$. A contradiction.

3) Eventually, once $ngroups = \frac{N}{groupsize}$, *ngroups* no longer changes.

Let $2^g = \frac{N}{groupsize}$. Because any neighbor of i at level g is also a neighbor of i at any level $k < g$, then by the PSS properties all neighbors of i at level g will be eventually added to is *localview*. These nodes will not be removed from i's *localview* (lines 5 to 7) while i is at any level $k \leq g$.

Once, by 2) i reaches level g and all its neighbors at level g belong to *localview* then i no longer merges (lines 8 to 10). And by 1), i never reaches any level larger than g. Therefore *ngroups* no longer changes and the node stabilizes.

Convergence. In order to validate the convergence of our algorithm we ran a simulation[1]. In this simulation we considered 10.240 nodes and a Peer Sampling Service that delivered messages with random references of nodes. Additionally, uniformly distributed position values were generated for every node and *groupsize* defined to 10. For this simulation in particular the correct number of groups (*ngroups*) is 1024.

At each simulation cycle a single PSS message was delivered to each node to be processed. The size of the PSS message influences directly the speed of convergence of the protocol. Typically, the message size increases logarithmically with the system size [Eugster et al., 2004]. We considered PSS messages containing 20, 30, 40, 50 or 100 node references[2]. In Figure 5.2 we depict the results of the simulations. The plot shows the percentage of nodes that hold a wrong estimation for *ngroups* per cycle.

From the results we can verify that the protocol converges to the desired configuration. It is also possible to see that, as expected, increasing the PSS message size improves the performance of the protocol. Nevertheless, note that it converges even for very small message sizes with respect to the size of the system.

Limitations. In this simplified version, the algorithm has also two important limitations. On one hand, if the desired group size does not exactly divide the number of nodes in the system in such way *ngroups* is a power of two, the algorithm does not stabilize. For instance, if in simulation of Figure 5.2 the number of nodes was 10.240 plus one the system would not stabilize completely. One group would detect an extra node in the system

[1]Code used for simulations is available at `github.com/fmaia/dataflasks_sim`

[2]Note that the protocol converges even with smaller message sizes however, considering the system size, smaller messages lead to slow convergence.

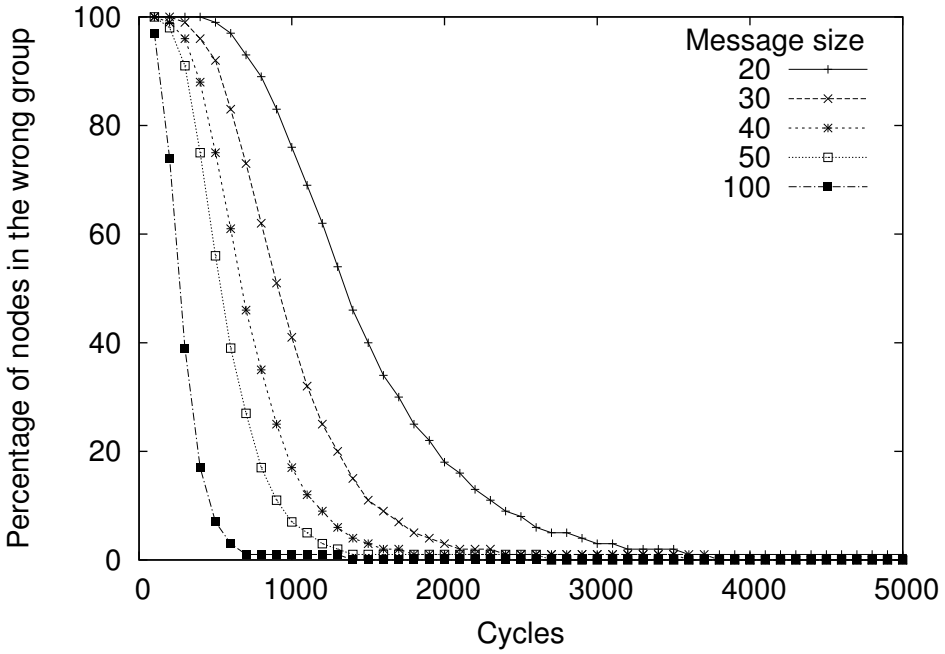


Figure 5.2: Convergence of 10.240 nodes running the simplified version of the group construction algorithm.

would not stabilize for any estimation of the number of groups. On the other hand, nothing is said about how it deals with churn. These two simplifications allow us to convey the main intuition behind the protocol in a straightforward way. Extensions to the basic protocol are proposed in Section 5.2 to solve both limitations.

5.2 Extensions

So far, the presented algorithm applies to a hypothetical system absolutely stable and with a round number of nodes. In practice however, the algorithm most probably never stabilizes but instead adapts to the dynamics of the membership. In this Section, we describe extensions to the algorithm of Section 5.1 in order to overcome the limitations identified previously. The first extension allows the protocol to support arbitrary system sizes. Next, we describe how the protocol can be extended to be able to handle churn.

```
   input    : min_groupize, max_groupsize, id
1  (...)
2  upon reception of m ← set of (id, pos) from PSS:
3      (...)
       /* need to merge or split?                                    */
4      if |locaview| < min_groupsize then
           /* Should Merge.                                          */
5          if ngroups > 1 then
6              ngroups ← ngroups/2
7      if |localview| > max_groupsize then
           /* Should Split.                                          */
8          ngroups ← ngroups * 2
9      (...)
```

Algorithm 14: Extended group construction algorithm.

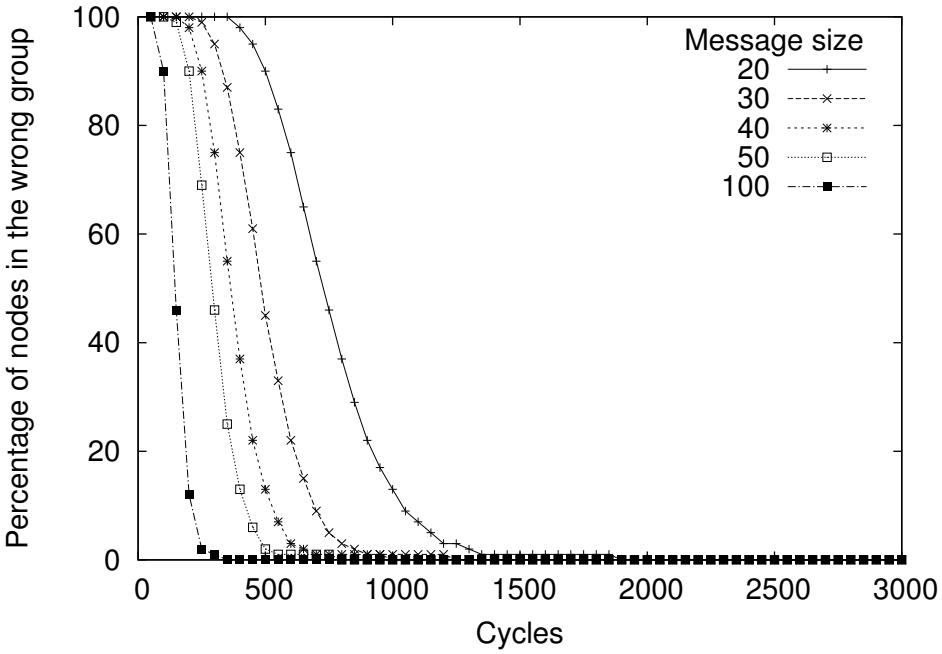
5.2.1 Handling arbitrary system sizes

As highlighted previously, the simplified algorithm presented in Section 5.1 is very sensitive with respect to the defined group size. The algorithm stabilizes if and only if there exists a power of two, *ngroups*, that exactly divides the system size in groups of *groupsize* nodes. The fact is that aiming at an exact group size (lines 8 to 12 in Algorithm 12) is restrictive. Moreover, for the type of systems we are considering knowing the exact size of the system is unfeasible. In order to tackle this limitation, we extend the algorithm in order to allow the definition of minimum and maximum group size thresholds. This simple enhancement allows the protocol to converge in real case scenarios. Algorithm 14 depicts the changes needed to add this extension.

In order to validate that the proposed extension does not impair the convergence of the protocol we conducted two simulations. In the first simulation, the conditions were similar to those of Section 5.1 for a system with 10,240 nodes and configured $min\_groupsize = 5$ and $max\_groupsize = 15$. As can be observed in Figure 5.3 the protocol converges preserving the desired behavior of the simplified version. Strikingly, the algorithm converges even faster when configured with this extension. This is due to the fact that splitting and merging decisions are delayed due to the threshold flexibility. Such delay allows the node to preserve more node references in its local view per cycle. These references allow better splitting and merging decisions speeding up the convergence process.

We then ran a second simulation. In this simulation we used the same configurations of the previous one but, this time, for a system with 15,000

Figure 5.3: Simulation of the flexible group size mechanism with 10.240 nodes.



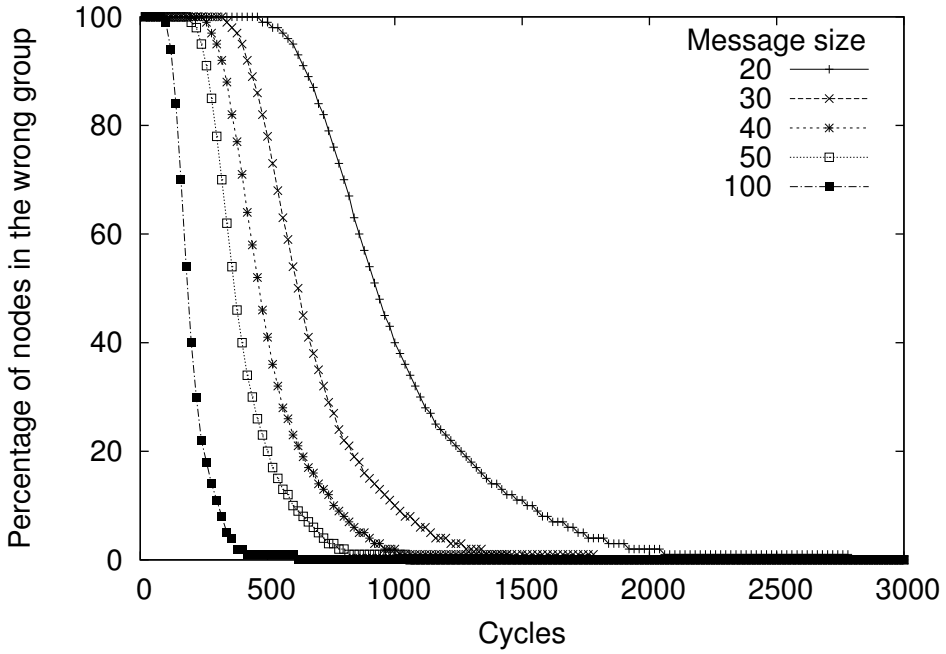
nodes. As observable in Figure 5.4, the protocol still converges to the desired configuration.

It is important to note that the maximum replication factor value must be at least double of the minimum. As the number of groups is always a power of two, choosing the replication thresholds this way avoids frontier cases where the system may enter a cycle of consecutive merge and split operations.

5.2.2 Handling churn

In order for the group construction algorithm to be useful it needs to be able to handle system dynamics. Nodes that leave the system must be eventually removed from every node's view while nodes entering the system must be incorporated. As stated previously, the group construction algorithm assumes the existence of a Peer Sampling Service. By design, the Peer Sampling Service is able to handle system dynamics. Nodes that leave the system are eventually removed from every sample the PSS delivers. Joining nodes, eventually, are sampled with the same probability as any other node in the system.

Figure 5.4: Simulation of the flexible group size mechanism with 15.000 nodes.



The problem of handling nodes joining the system is immediately solved. New nodes are eventually sampled by the PSS and incorporated into the group construction algorithm. However, as it is, the group construction algorithm is unable to handle node departure. In fact, even if not subsequently sampled by the PSS, departure nodes for which there is a reference stored in a node view are never *forgotten* if they belong to the same group as the node. This limitation can be overcome introducing an aging mechanism for node references. Each node reference is now tagged with an age property. When a node reference is delivered by the PSS it is tagged with age 0. Each time a new PSS message is delivered, every node sees its age increased by 1. With this extension, it is now possible to define a maximum age threshold to allow node references to be forgotten when obsolete. Note that, if a node leaves the system eventually ceases to be sampled by the PSS. Consequently, every of its node references stored in any of the active nodes will inevitably age beyond the age threshold and eventually be forgotten. As a result, the protocol is now able to handle node departure.

Nevertheless, it is necessary to define an adequate maximum age threshold. Intuitively, such threshold must be higher than the number of cycles required to refresh a reference to a valid node. Note that, if this was not

the case, valid references would be continuously removed impairing the convergence of the algorithm. In practice this means defining the age threshold superior to the time the PSS needs to sample a certain node reference. Unfortunately, this is too slow. As the system size grows, the probability of a certain node to be sampled by the PSS in each cycle decreases. Consequently, the number of cycles needed to make sure a certain node is sampled becomes unmanageably high.

In DataFlasks, we tackle this limitation by adding an active thread to the group construction algorithm. Periodically, each node produces a node reference to itself with age 0. It then sends such reference, alongside with all the references in the local view, to all the nodes it estimates to be in the same group. This simple mechanism, allows refreshing node references. Note that, once a node has left the system, every reference to it that may exist in the system will stop being refreshed. Eventually, it is removed from every node's local view as desirable. The active thread may be seen as an *heart beat* mechanism. An important thing to notice is that, although this mechanism is not required for convergence, it improves significantly the algorithm's speed of convergence. As nodes exchange references with nodes from the same group periodically, nodes receive useful references without waiting for the PSS to sample them all. Note that, as described in [Voulgaris and van Steen, 2013], the Peer Sampling Service randomness properties are essential but, typically, not sufficient to achieve good convergence results.

Figure 5.5, depicts the results for a simulation of the group construction algorithm with the all the extensions described so far. The simulation was configured with 15,000 nodes, $min\_groupsize = 5$, $max\_groupsize = 15$ and different view sizes. Additionally, the maximum age threshold was defined to 30 and the active thread is launched every 15 cycles. As observable, the algorithm still converges to the desired organization and is now much faster.

It remains to access the actual ability of the algorithm to reconfigure itself when there is a massive departure or entrance of nodes. Note that, with the extension considered for handling arbitrary system sizes, moderate system dynamics are inherently handled. In order to force system reconfiguration it is necessary that the number of nodes entering or leaving the system be sufficient to force a split or merge operation respectively. In Figures 5.6(a) and 5.6, we depict the results for two simulations. Figure 5.6(a) presents the results for an experiment where 7,500 nodes are added to a stable system of 7,500 nodes. Such membership change is made at cycle 500. The line in the Figure represents the percentage of nodes with a wrong estimation for the number of groups. Note that the expected estimation changes at cycle 500. In this case, the configuration dictates that initially the correct number of groups is 512. With the addition of another 7,500 nodes, the number of

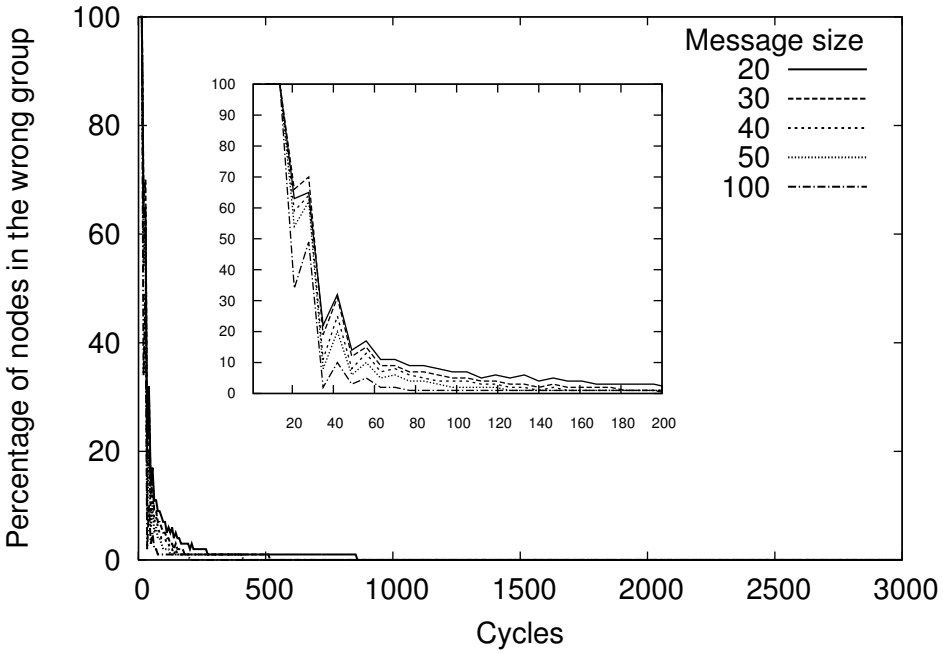


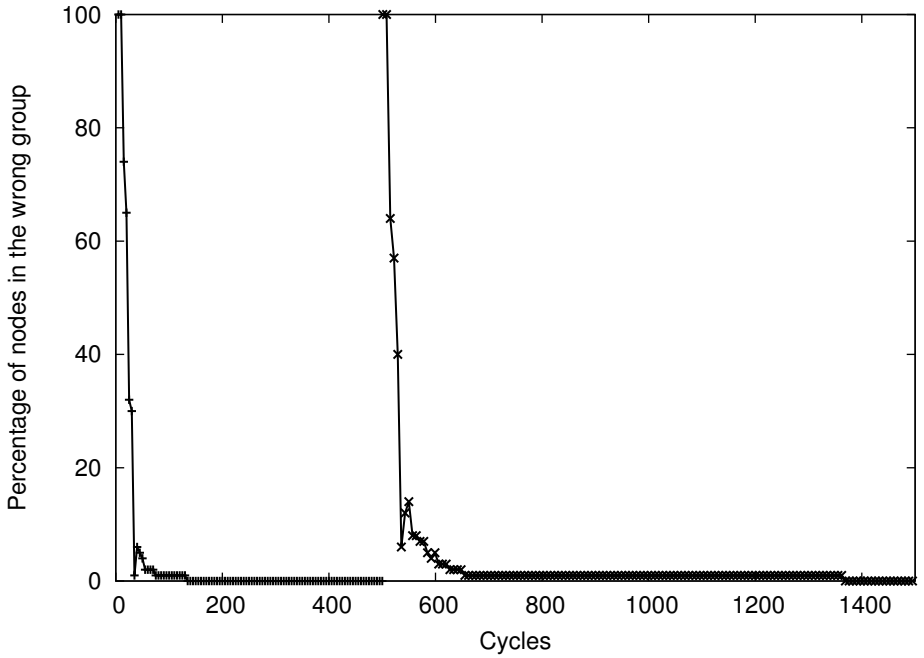
Figure 5.5: Convergence of 15.000 nodes running the extended version of the group construction algorithm.

groups is expected to be 1,024. Consequently, until cycle 500, the line depicts the percentage of nodes that do not estimate the number of groups to be 512, after such cycle it depicts the percentage of nodes that do not estimate it to be 1,024. As it is observable, after the abrupt addition of nodes the algorithm is able to converge to the new configuration.

In an analogous experiment, 50% of the nodes were removed from a 15,000 node stabilized system. The results are depicted in Figure 5.6. In this case, until cycle 500 the plot depicts the percentage of nodes that have a number of groups estimation different from the 1,024. After cycle 500, it depicts the percentage of estimations that are not 512. In fact, the removal of 50% of the system nodes forces the algorithm to perform a merge operation in order to preserve the replication factor above the minimal threshold value. Similarly to the previous experiment, the algorithm is able to converge as desired.

5.3 Discussion

In this chapter we presented a new algorithm for peer-to-peer group construction. It is designed as an unstructured, proactive gossip-based protocol



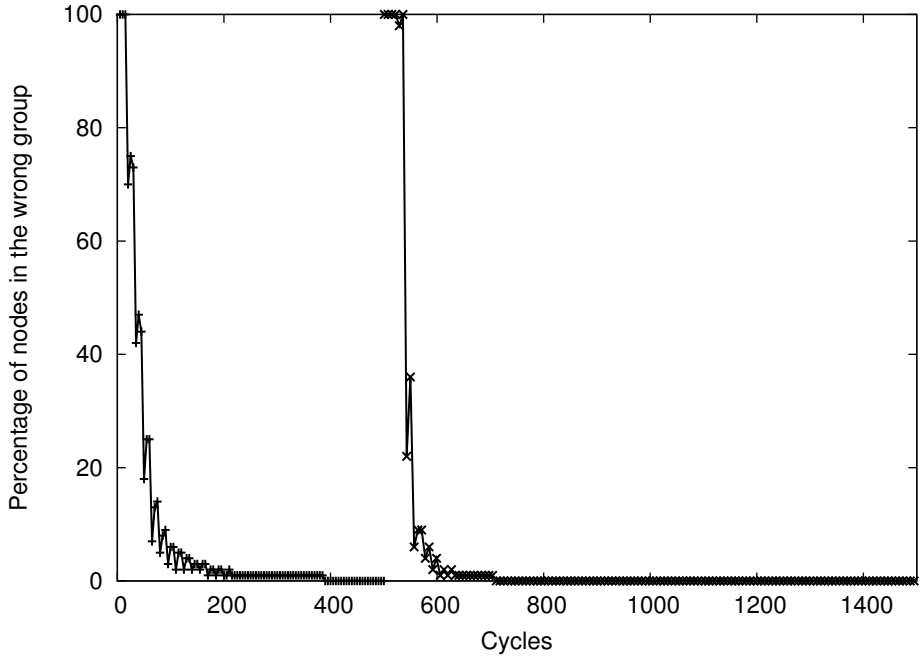
(a) Simulation of 7,500 nodes. Number of nodes doubled at cycle 500.

solely based on a Peer Sampling Service. As a result it is inherently scalable and resilient to system dynamism. We have shown that it is able to cope with dramatic system failures and able to be deployed in several thousands of nodes. Moreover, we provide a proof of convergence for the algorithm.

The design of the group construction algorithm is heavily influenced by the fact it is used as a core component for DataFlasks. Namely, considering a total number of groups to always be a power of 2 reduces data transfers whenever there is a configuration change. Such choice renders the design ideal in that context. However, the protocol is not dependent on DataFlasks. It can be used separately as a building block for other applications and it is a contribution in itself.

Finally, there are some considerations to be made regarding the protocol assumptions. The first important assumption concerns the node *position*. Each node must generate a position value in the interval $]0, 1]$ uniformly at random across the entire system. This position is used to balance nodes by groups. Naturally, good load balancing is only expected with high probability for a very large number of nodes. This must be taken into account when using our protocol. The second important assumption is about churn and faults. Churn and faults are also considered to be balanced across the entire system. If for some reason, faults and churn are biased, for instance, to a specific

Figure 5.6: Simulation of 15,000 nodes. At cycle 500, 7,500 nodes are removed from the simulation.



group of nodes the protocol may struggle to continue working properly. The easiest way to think about this is considering a system were churn is as high as 25%. If, for some reason, when 25% of the nodes are failing, an entire group is among those 25% of nodes then data belonging to such group will be lost. Even so, we believe that assuming churn and faults balanced across the entire system to be reasonable.

Chapter 6

Proof of concept

Having described the design and architecture of DataFlasks and the protocols that support its implementation, we now focus on providing a proof of concept for our system. To this end we implemented a complete DataFlasks prototype in Java. In this chapter we describe the implementation of the DataFlasks prototype detailing the different components and present an experiment that validates our approach.

6.1 DataFlasks Prototype

Our aim with system prototype was to offer the possibility of a real assessment of the DataFlasks system, complementing simulation results. However, we do not have access to a sufficiently large number of machines allowing us to run DataFlasks in the large scale environment it was designed for. We opted by a compromise between a real deployment and a pure simulation environment.

The Minha framework [Carvalho et al., 2011] provides Java developers with the possibility of running several instances of their applications in a single Java Virtual Machine. The framework is able to provide a simulated environment and give applications the illusion of running in their own JVM and machine. The code that runs on Minha is unmodified real, ready for production deployment and that can be evaluated as if it was running on thousands of machines.

We have implemented DataFlasks leveraging the Minha framework. The DataFlasks prototype is open sourced and available at `github.com/fmaia/dataflasks`. Its architecture follows the one depicted in Figure 6.1.

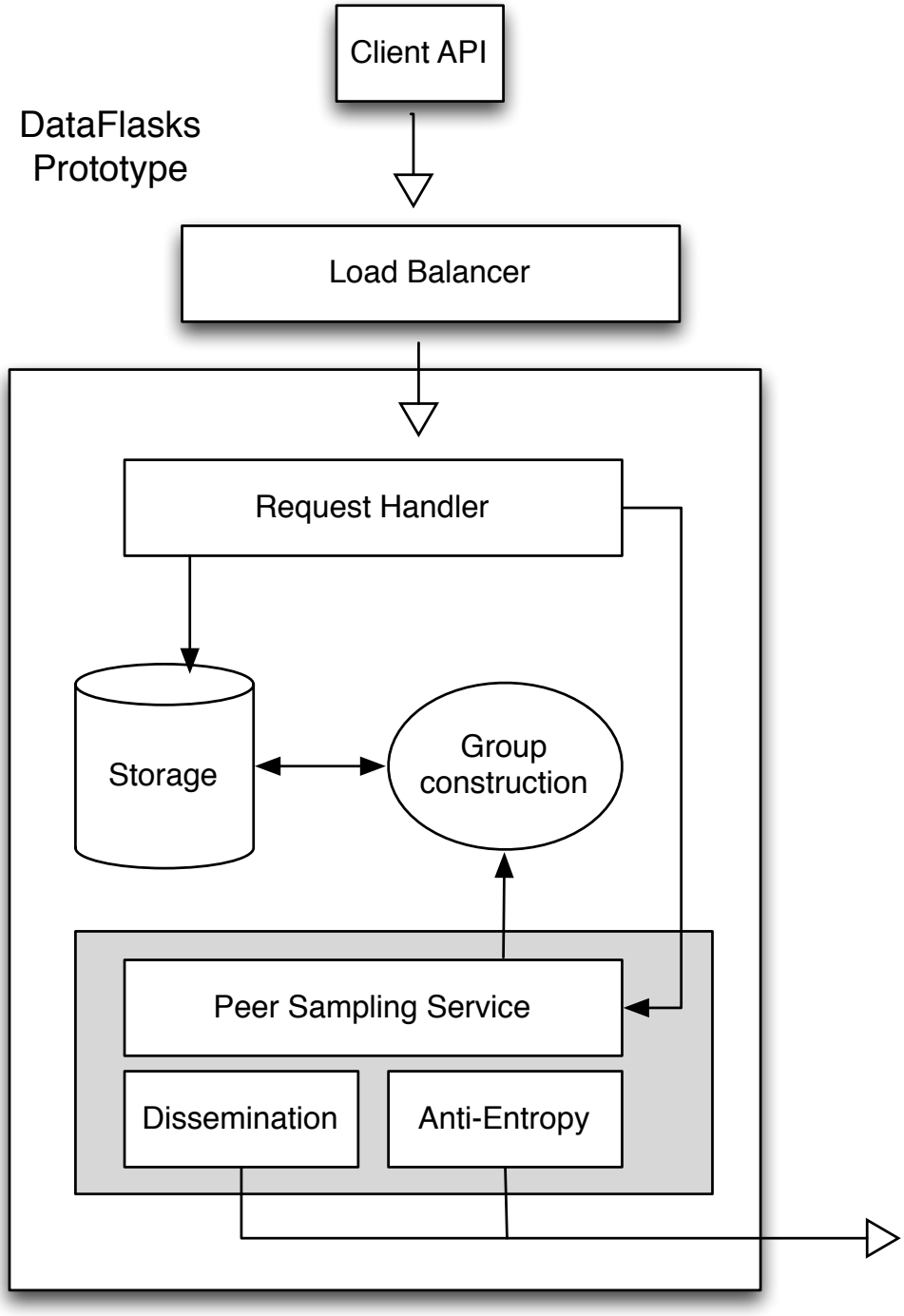


Figure 6.1: Dataflasks prototype overview architecture.

Nodes communicate through UDP sockets and the current prototype assumes that all nodes are in the same network and are able to contact each other knowing only their IP addresses. Problems arising from removing this assumption are out of the scope of the present dissertation. Next, we describe the role of the prototype components and their implementation.

6.1.1 Node Communication

At the base of our prototype is the node communication component. In Figure 6.1, this component is represented by a gray area containing three other sub-components: the Peer Sampling Service, the Dissemination component and the Anti-entropy component.

Firstly, we have implemented a version of the Cyclon protocol [Voulgaris et al., 2005a] that serves as our Peer Sampling Service. This service provides each DataFlasks node with a continuous stream of random peers. It maintains the node's view of the system and allows node discovery as described in previous chapters. Our implementation of the Cyclon protocol includes a boot component that initiates the Cyclon protocol. Even though this is a simplification, it doesn't affect the behavior of DataFlasks. It replaces the node joining procedure described in [Voulgaris et al., 2005a] by providing an initial random sets of peers to populate the node's initial view.

For data dissemination, the current implementation of DataFlasks leverages the Peer Sampling Service. Each request is disseminated in an *infect and die* model [Eugster et al., 2004]. To this end, each node forwards each request to the peers in their Cyclon view. The size of the view is considered to be large enough to provide good dissemination properties as described in [Eugster et al., 2004]. Note that, requests are not required to reach all nodes in the system. It is sufficient to ensure it reaches some nodes of each group. Once a request reaches a node belonging to the group of interest, such node disseminates it to the nodes in its group. This allows for optimizations, some of them already implemented in our prototype.

Finally, in order to ensure data reaches all nodes that must replicate it, DataFlasks uses an anti-entropy mechanism [Demers et al., 1987]. The mechanism works by, periodically, have nodes contact another random node in its group and exchange information about the data they are replicating. If a node detects that it lacks some data, i.e., it detects it is not replicating certain *(key,object)* pairs, it asks other group nodes for them. This mechanism allows nodes to periodically restore possibly missing data items. Moreover, it ensures that node entering the system are automatically integrated. In detail, when a node enters the system starts its own anti-entropy mechanism. At that time, its local storage is still empty thus the first anti-entropy cycles

allow the node to quickly receive the data it is responsible for storing.

6.1.2 Client Interface and Load Balancing

DataFlasks offers a client interface package, which abstracts all interactions with the data store. This package offers *put* and *get* methods to the client application. By design, DataFlasks supports multiple versions for the same object. However, as it does not enforce consistency of write operations, the pair (key,version) is considered unique. Consequently, the client interface generates a unique key per data object obtained by concatenating the pair (key,version) and encoding it in an unique long value. Data objects are arrays of bytes and the application is responsible for encoding its data appropriately. Consequently, a *put* operation receives as input the object key and an object and a *get* operation returns an object correspondent to a key given as input.

The client interface package has one important configuration parameter. When the client application issues a request it can define the number of replies to wait for. Recall that in DataFlasks requests are disseminated through the system and several nodes will answer them. Specially for write operations, it is useful to wait for a specific number of nodes to reply. This way, the client application can define a number of replicas of the object it believes are sufficient to ensure data persistence. In fact, even though with very high probability requests will arrive to all the other node replicas, waiting for a certain number of answers increases client trust in the persistence of data.

Another important aspect of the client interface package is node discovery. In DataFlasks, any node may receive requests but the client interface package must be able to discover them in order to issue data requests. This functionality is provided by a Load Balancer component. Similarly to the Peer Sampling Service boot mechanism described previously, in the current implementation, the load balancer is implemented artificially leveraging the Minha platform. In a real deployment, the load balancer should behave as an observer of the Peer Sampling Service, which provides a node discovery mechanism. Moreover, the current implementation of the load balancer provides random node samples from the entire system for each request. This is sufficient for our proof of concept but clearly yields suboptimal performance. Many optimizations are possible. The most simple one would be caching. Once a node has satisfied a request for a certain key, i.e. it belongs to a certain group, it may be stored with such information. Subsequent requests can then be judiciously routed to appropriate nodes significantly improving performance.

For our experiments, we implemented a YCSB [Cooper et al., 2010] bind-

ing for DataFlasks. Our binding supports *get* and *put* operations. Even though YCSB supports *update* and *delete* operations, these are not currently supported by DataFlasks. Once DataFlasks does not enforce data consistency, it is not possible to implement the *update* operation with well-defined semantics. Instead, in order to update a data object, the client application should issue a *put* operation with a new version for such object. On the other hand, the *delete* operation can be implemented in DataFlasks in a similar way to the *put* operation. Additionally, the client interface package was implemented with support for multi-threaded client applications. As a consequence, YCSB can be run in multi-threaded mode.

6.1.3 Group Construction

The group construction component includes the algorithm described in Chapter 5 with all the extensions of Section 5.2. The Cyclon implementation was enhanced to allow the propagation of node's position alongside the node's identification. This way, each time the Peer Sampling Service component receives peer references, not only processes them but also delivers them to the group construction component. At the reception of peer references, the group construction algorithm progresses as described in Chapter 5.

This component continuously exports its estimation of the number of groups in the system and the group to which the node belongs to. This information is used by the storage component in the decisions of storing or discarding data as described previously.

6.1.4 Storage

The storage component is currently an in-memory store. Our prototype is modular and the store can be replaced with a disk based one, however, because in our evaluation we consider that when a node fails all data is lost, an in-memory store is adequate for our evaluation purposes. The storage offers, internally, the same write and read operations as the system as a whole. When a read operation is issued the storage checks if it holds the data corresponding to the requested key. If that is the case, it replies with the data object. Otherwise, it replies with an empty object. For the write operation, the storage component asks for the group the node belongs and decides if it is responsible for the data object being written or not (recall Algorithm 11). It replies with a boolean object saying if it has stored or not the object.

6.1.5 Request Handler

The request handler is implemented as a typical multi-threaded server. Each request is handled by thread and this thread leads the request through the steps necessary to process it. Two main operations are implemented: the *get* operation and the *put* operation. Algorithm 15 presents their pseudo-code.

```
1  upon reception of r ← request from cli ← sender:
       Data: rlog ← []/* Read request log.                              */
       Data: wlog ← []/* Write request log.                             */
       Data: op ← r.type
2      if op is get then
           Data: key ← r.key
           Data: requestID ← r.id
3          if requestID in rlog then
               /* Ignore operation because it was seen previously.      */
4          else
5              rlog.add(requestID)/* Mark as seen.                     */
               Data: data ← storage.get(key)
6              if data then
                   /* Locally available.                               */
7                  replyToClient(cli, data)
8              else
9                  forwardRequest(r)
10     else if op is put then
           Data: key ← r.key
           Data: object ← r.object
11         if key in wlog then
               /* Ignore operation because it was seen previously.      */
12         else
13             wlog.add(key)/* Mark key as seen.                        */
14             stored ← storage.put(key, object)
15             if stored then
16                 replyToClient(cli)
17             forwardRequest(r)
```

Algorithm 15: Pseudo-code for the Request Handler component.

There are a few things to notice in the processing of these operations. Firstly, read operations need a mechanism to detect duplicate requests. Since requests are being disseminated in an epidemic way, they can arrive at each node more than once. If these duplicates are not detected, nodes will continuously forward them and they will never stop being in transit in the system. This is achieved by uniquely identifying read requests. In our prototype,

client applications are also given a unique id, similar to the one given to DataFlasks nodes. With this identification and a request counter maintained by the client interface component, requests are tagged with a request id. In Algorithm 15, this mechanism is described with the use of a *read log (rlog)*. Upon reception of a read request, its id is compared against the information in the log in order to detect if it was already processed (lines 3 and 5). In that case the request is ignored.

A similar mechanism exists for write requests using a *write log (wlog)* (lines 10 and 12). However, for write requests there is no need for a unique request id as the request key is sufficient to detect possible duplicates. In fact, while a read request may be issued several times for the same key, a put request is always the same for a certain key. This is true because DataFlasks assumes that write consistency is, currently, the responsibility of the client application.

Secondly, the mechanism for request forwarding (lines 8 and 16) can be also improved. In the default configuration, requests are disseminated using the Peer Sampling Service view, which is a random set of other nodes in the system. However, if the node detects that the request is aimed at his own group it can leverage the information it has about his group peers and forward the request to them directly. This mechanism reduces system load and accelerates the request processing and it is implemented in our prototype.

Finally, the *ReplyToClient* operation (lines 7 and 15) is responsible for replying to the client application when the operation succeeded at a node. For the read operation it replies with the requested data while, for the write operation, simply confirms that the data was stored in that node. An important implementation detail concerns a parameter that can define a probability of actually replying to the client. In the normal setting, every node that can satisfy the request replies to the client. When DataFlasks groups grow in size, several answers are sent to the client, which can unnecessarily overload it. In order to tackle this problem, in our implementation, it is possible to define a probability of answering to the client smaller than one. When set to 0.7 for example, allows that only 70% of the group nodes actually issue answers. This mechanism leverages the assumption that nodes have access to a random number generator thus this choice can be performed locally.

6.2 Experiments

Having described our DataFlasks prototype, we present a set of experiments to validate persistence guarantees of our system. The goal of DataFlasks was to be highly scalable and resilient under high levels of churn.

We focus on these goals and leave performance evaluation out of the scope of the present work. We used a machine with an AMD Opteron 6172 (24 core at 2.1GHz) and 128GB of memory. Leveraging Minha, we are able to evaluate DataFlasks in a large scale environment while, at the same time, provide a Java prototype that is actually ready for deployment in a real scenario.

In our experiments we ran 1000 nodes and populated DataFlasks with 200.000 data objects. We configured the Peer Sampling Service to exchange messages every 2 seconds and the active group construction mechanism (see Section 5.2.2) every 15 seconds. Additionally, the anti-entropy mechanism was configured to run at intervals of 30 seconds.

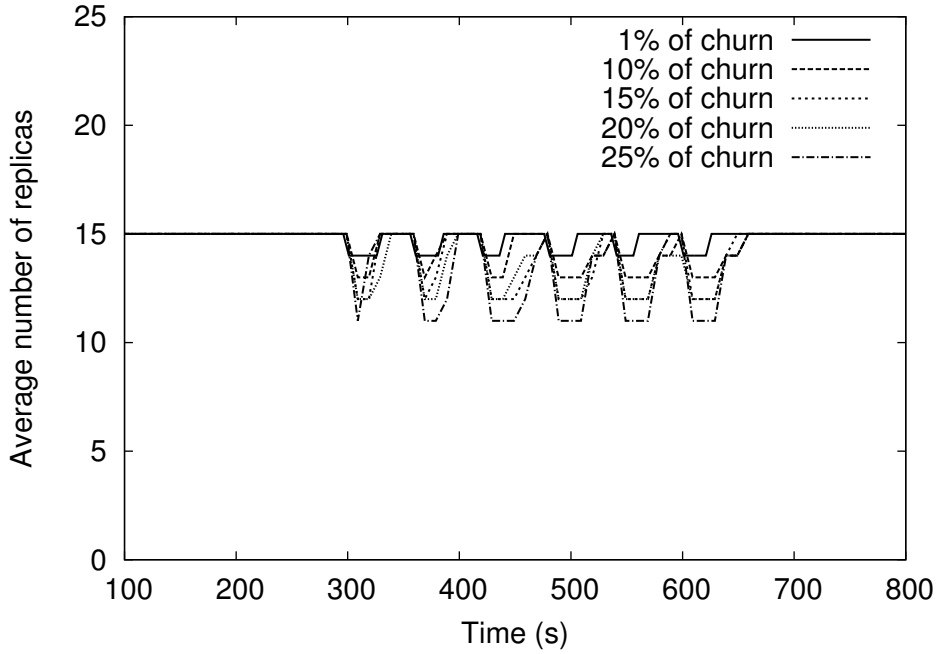


Figure 6.2: DataFlasks behavior for different levels of churn.

We subjected the system to different levels of churn and recorded the number of replicas per key each 10 seconds. Churn is implemented by removing a node and adding a fresh one preserving the position distribution of the nodes removed. We assume that churn is uniform across the entire system, which in a very large scale scenario means that nodes leaving and entering are uniformly distributed by all groups. Consequently, in our experiments, subjecting the system to 10% of churn means subjecting each group to 10% of churn. Moreover, churn is applied each 60 seconds for 5 minutes. The results of the experiments are depicted in Figure 6.2.

As it is observable, during the churn phase (between cycles 300 and 600),

the average number of replicas is not constant as it is impaired by the departure of nodes. However, DataFlasks is able to repair the number of replicas even for churn as high as 25%. In all the experiments no data was lost by DataFlasks.

6.3 Discussion

We have shown that DataFlasks design achieves the goals initially set. DataFlasks has an elegant design that effortlessly scales being able to be deployed in large scale scenarios. Moreover, it is able to handle high levels of node churn.

In order to better assess DataFlasks, we wanted to compare its behavior with that of an existing system. However, this was not possible. Notably, it was not possible to compare with existing data stores such as Cassandra [Lakshman and Malik, 2010] or similar systems as the use of 'one-hop' DHT is not possible with our assumption that no node can know every other node in the system. Following this observation, we thought of comparing directly the behavior of the core of DataFlasks with an existing implementation of a 'multi-hop' DHT system, which are at the core of systems such as PAST [Druschel and Rowstron, 2001] or OceanStore [Kubiatowicz et al., 2000]. Accordingly, we tried to run an experiment with OpenChord (`http://sourceforge.net/projects/open-chord/`), the most up-to-date Java implementation of the Chord protocol we found. However, in our experiments, both in a real deployment and in Minha, a simple sequence of single node departures and joins lead to system deadlocks. The rigid lookup structure of Chord seems to effectively impair its capacity to handle churn. As a consequence, we searched for an implementation of the Kademlia protocol, which is known to be able to handle high amounts of churn [Wang and Kangasharju, 2013] and would be more adequate for a behavior comparison. Moreover, some of the characteristics of Kademlia's design resemble that of DataFlasks'. In particular, Kademlia buckets can be seen as DataFlasks groups and both systems assume that nodes have access to a uniformly distributed value. The two approaches differ in how node discovery is achieved, in how data replication is maintained and in the overlay used for information lookup. We found two candidate implementations: OpenKad (`https://code.google.com/p/openkad/source/list`) and TomP2P (`http://tomp2p.net`). Unfortunately, OpenKad only implements DHT lookup and message passing operations and could not be used to compare data persistence under churn. TomP2P on the other hand is a mature peer-to-peer framework and implements a panoply of peer-to-peer

mechanisms. Because this framework is so complex we were not able to run it in the Minha framework in a timely fashion. At a scale similar to the one used in DataFlasks experiments, the simulation became impracticably slow.

Finally, we left performance evaluation out of the scope of the present work. Our focus was on how DataFlasks handles high dynamism and an adequate performance evaluation requires maturation of some DataFlasks components that were not the focus of the dissertation. In particular, a comprehensive study of the more adequate epidemic dissemination protocol for DataFlasks is necessary as well as a careful design and implementation of the DataFlasks Load Balancer component. Even so, DataFlasks successfully achieved our scalability and resilience goals and proved to be suitable to environments with very high levels of churn.

Chapter 7

Conclusion

With the ever-increasing number of connected devices and the ubiquity of the Internet, systems composed of several thousand of peers are becoming a commonplace. Taking advantage of all these devices and having them cooperating to provide the next generation of services and applications demands distributed protocols able to handle unprecedentedly challenging environments. Along this dissertation work we looked at one of the pivotal challenges of very large scale systems: dynamism. We believe that designing distributed systems able to cope with constant churn and failures is the only way to guarantee that we are going to be able to address the challenges of future information systems.

We have striven for the design of a new generation of data management systems. Designed from the ground up to be able to deal with an exponentially increasing volume of data as well as an unprecedented dynamic environment. The presented data store, DataFlasks, can be easily scaled to several thousands of nodes. It was designed to be completely decentralized and to be equipped with mechanisms that are able to overcome high and constant membership dynamism due to churn or failures.

The work on DataFlasks resulted in the proposal of two novel epidemic protocols aimed at dividing an arbitrary large system into groups. Both protocols were also designed for large scale environments and, individually, also exhibit the desirable scalability and resilience characteristics of DataFlasks. They were carefully evaluated and not only serve as a key component in the architecture of DataFlasks but also as an independent and general purpose building block for other applications. In particular, in mobile environments or in Internet-scale systems it is of vital importance to be able to organize nodes even when they are continuously entering or leaving the system. We believe that DataFlasks and the proposed protocols are part of a new generation of large scale systems that pro-actively

repair themselves and are able to autonomously adapt themselves to dynamic environments.

In this dissertation we also presented a proof of concept of DataFlasks. We focused on how DataFlasks was able to handle high dynamism and left performance concerns out of the scope of the present work.

Following the work on DataFlasks, some interesting research paths were identified. We focus on two. Firstly, DataFlasks leaves data consistency concerns to the client application. While a compromise between data consistency and effective dynamism handling must always be made, we believe that DataFlasks design paves the way for a useful coexistence of different compromise levels. In previous work [Maia et al., 2011; Pereira and Oliveira, 2004], we showed that it is possible to use epidemic style communication patterns to achieve coordination between hundreds of nodes spread around the world. Leveraging such work and integrating it with DataFlasks can lead to new and practical data consistency levels. Moreover, we believe it is possible to adapt the protocol proposed in [Pereira and Oliveira, 2004] and take its epidemic-style design influence a step further. In particular, investigate if it is possible to relax the guarantees offered by the protocol, still offering a level of guarantees easily understandable and usable by application developers and, at the same time, offer the dynamism handling of epidemic protocols. Secondly, from the perspective of the group construction algorithm of DataFlasks we plan on investigate if the assumption that nodes are uniformly distributed can be relaxed while still achieve autonomous and completely distributed behavior. This assumption eases the process of node-local, independent decisions because it is, in fact, implicit knowledge. It is important to investigate if this assumption can be relaxed or it is actually mandatory to achieve DataFlasks-type behavior.

Bibliography

Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007. (Cited on page 39.)

David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002. (Cited on page 13.)

Norman TJ Bailey et al. *The mathematical theory of infectious diseases and its applications*. Charles Griffin & Company Ltd, 5a Crendon Street, High Wycombe, Bucks HP13 6LE., 1975. (Cited on page 14.)

Ranjita Bhagwan, Stefan Savage, and Geoffrey M Voelker. Understanding availability. *International Workshop on Peer-to-Peer Systems*, pages 256–267, 2003. (Cited on page 24.)

Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (Cited on pages 26, 37, 38, 39, 40, 47, and 49.)

Nuno Carvalho, Jose Pereira, Rui Oliveira, and Luis Rodrigues. Emergent structure in unstructured epidemic multicast. *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 481–490, 2007. (Cited on page 54.)

Nuno A. Carvalho, João Bordalo, Filipe Campos, and José Pereira. Experimental evaluation of distributed middleware with a virtualized java environment. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, 2011. (Cited on page 75.)

Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007. (Cited on page 24.)

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008. (Cited on pages 1, 2, 11, and 12.)

Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, 2003. doi: 10.1145/863955.864000. URL `http://doi.acm.org/10.1145/863955.864000`. (Cited on page 13.)

Kai Cheng, Limin Xiang, and Mizuho Iwaihara. Time-decaying bloom filters for data streams with skewed distributions. *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA)*, pages 63–69, 2005. (Cited on page 49.)

Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. *Designing Privacy Enhancing Technologies*, pages 46–66, 2001. (Cited on page 13.)

E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, February 1982. ISSN 0001-0782. doi: 10.1145/358396.358400. URL `http://doi.acm.org/10.1145/358396.358400`. (Cited on page 10.)

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing*. ACM, 2010. (Cited on page 78.)

Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008. (Cited on pages 2, 11, 12, and 13.)

Jeff Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009. (Cited on page 1.)

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian,

Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007. (Cited on pages 2, 3, 11, 13, and 24.)

Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987. (Cited on pages 14, 21, and 77.)

P. Druschel and A. Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001. (Cited on pages 11 and 83.)

P Erdős and A Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi, vol*, 2:482–525, 1976. (Cited on page 16.)

P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003. (Cited on pages 15, 21, and 27.)

P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *Computer*, 2004. (Cited on pages 15, 21, 65, and 77.)

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000. (Cited on pages 48 and 49.)

Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Etienne Rivière, and Spyros Voulgaris. Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams. In *Peer-to-Peer Networking and Applications*. Springer US, 2012. (Cited on page 21.)

Antonio Fernández, Vincent Gramoli, Ernesto Jiménez, A-M Kermarrec, and M Rayna. Distributed slicing in dynamic systems. *27th International Conference on Distributed Computing Systems (ICDCS)*, pages 66–66, 2007. (Cited on pages xv, 14, 16, 21, 23, 25, 26, 27, 28, 34, 42, 44, 45, and 54.)

Ayalvadi J Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. *International COST264 Workshop on Networked Group Communication*, pages 44–55, 2001. (Cited on pages 14 and 23.)

John Gantz and David Reinsel. Extracting value from chaos. *IDC iview*, pages 1–12, 2011. (Cited on page 1.)

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. (Cited on pages 2 and 11.)

Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, and Robbert van Renesse. A fast distributed slicing algorithm. *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 427–427, 2008. (Cited on pages xv, 14, 16, 24, 25, 26, 27, 29, 34, 42, 44, 45, and 54.)

Vincent Gramoli, Ymir Vigfusson, Ken Birman, A-M Kermarrec, and Robbert Van Renesse. Slicing distributed systems. *IEEE Transactions on Computers*, 58(11):1444–1455, 2009. (Cited on pages 21, 23, 25, 26, 27, 30, 44, and 45.)

Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. *ACM SIGMOD Record*, 25(2):173–182, 1996. (Cited on pages 2 and 11.)

R. Guerraoui, R. Oliveira, and A. Schiper. Stubborn Communication Channels. Technical report, EPFL, 1998. (Cited on page 61.)

Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. *Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 1 – 6, February 2006. (Cited on page 24.)

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. *NSDI*, 2004. (Cited on page 13.)

Richard Guy, Peter Reiher, D Rather, Michial Gunter, Wilkie Ma, and Gerald Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. *Lecture notes in computer science*, pages 254–265, 1999. (Cited on page 10.)

Richard G Guy, John S Heidemann, Wai-Kei Mak, Thomas W Page Jr, Gerald J Popek, Dieter Rothmeier, et al. Implementation of the ficus replicated file system. *Summer*, pages 63–72, 1990. (Cited on page 10.)

Xiaojun Hei, Chao Liang, Jian Liang, Yong Liu, and Keith W Ross. A measurement study of a large-scale p2p iptv system. *IEEE Transactions on Multimedia*, 9(8):1672–1687, 2007. (Cited on page 24.)

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of lua. *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1, 2007. (Cited on page 51.)

Mark Jelasity and A-M Kermarrec. Ordered slicing of very large-scale overlay networks. *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 117–124, 2006. (Cited on pages 51, 54, and 56.)

Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 79–98, 2004. (Cited on page 60.)

Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007. (Cited on pages 20, 25, 26, 27, 30, 42, and 60.)

Paulo Jesus, Carlos Baquero, and Paulo Sergio Almeida. Fault-tolerant aggregation for dynamic networks. *29th IEEE Symposium on Reliable Distributed Systems*, pages 37–43, 2010. (Cited on pages 14 and 24.)

Anne-Marie Kermarrec and Maarten Van Steen. Gossiping in distributed systems. In *ACM SIGOPS Operating Systems Review*. ACM, 2007. (Cited on page 21.)

Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. In *IEEE Transactions on Parallel Distributed Systems*. IEEE, 2003. (Cited on page 21.)

Rusty Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010. (Cited on page 11.)

Jeff Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, 2007. (Cited on page 1.)

John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Notices*, 2000. (Cited on pages 12 and 83.)

Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2): 35–40, 2010. (Cited on pages 2, 3, 11, 13, 24, and 83.)

Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43 (2):12–14, 2010. (Cited on pages 2 and 12.)

Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). 9:185–198, 2009. (Cited on page 51.)

Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware based data replication providing snapshot isolation. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, 2005. (Cited on pages 2 and 11.)

F. Maia, M. Matos, E. Rivière, and R. Oliveira. Slicing as a distributed systems primitive. *6th Latin-American Symposium on Dependable Computing*, 2013a. (Cited on page 14.)

Francisco Maia, Miguel Matos, José Pereira, and Rui Oliveira. Worldwide consensus. *Distributed Applications and Interoperable Systems*, pages 257–269, 2011. (Cited on pages 14, 24, and 86.)

Francisco Maia, Miguel Matos, Etienne Rivière, and Rui Oliveira. Slead: low-memory, steady distributed systems slicing. *Distributed Applications and Interoperable Systems*, pages 1–15, 2012. (Cited on pages 14, 42, 44, 46, 47, 49, 54, and 56.)

Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira, and Etienne Rivière. Dataflasks: An epidemic dependable key-value substrate. *In the International Workshop on Dependability of Clouds, Data Cen- ters and Virtual Computing Environments - Dependable Systems and Networks Workshops, (DSN-W)*, 2013b. (Cited on page 23.)

Francesco Marchioni. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012. (Cited on page 11.)

Miguel Matos, Valerio Schiavoni, Etienne Rivière, Pascal Felber, and Rui Oliveira. Laystream: composing standard gossip protocols for live video streaming. *The International Conference on Peer-to-Peer Computing*, 2014. (Cited on page 14.)

Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, 2002. (Cited on page 14.)

Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. *IEEE Ninth International Conference on Peer-to-Peer Computing (P2P)*, pages 99–100, 2009. (Cited on page 30.)

Alberto Montresor, Márk Jelasity, and Ozalp Babaoglu. Decentralized ranking in large-scale overlay networks. *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 208–213, 2008. (Cited on pages 21 and 23.)

Napster. www.napster.com. (Cited on page 13.)

Mathieu Pasquet, Francisco Maia, Etienne Rivière, and Valerio Schiavoni. Autonomous multi-dimensional slicing for large-scale distributed systems. *Distributed Applications and Interoperable Systems*, pages 141–155, 2014. (Cited on page 57.)

J Pereira and R Oliveira. The mutable consensus protocol. *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, 2004. (Cited on page 86.)

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001. ISSN 0146-4833. doi: 10.1145/964723.383072. URL `http://doi.acm.org/10.1145/964723.383072`. (Cited on page 13.)

Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. *Proceedings of the USENIX Annual Technical Conference*, pages 127–140, 2004. (Cited on pages 3, 13, and 19.)

Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. *Proceedings of the First International Conference on Peer-to-Peer Computing*, pages 99–100, 2001. (Cited on page 13.)

Etienne Rivière and Spyros Voulgaris. Gossip-based networking for internet-scale distributed systems. *E-Technologies: Transformation in a Connected World, Lecture Notes in Business Information Processing*, 78:253–284, 2011. (Cited on pages 14 and 23.)

Étienne Rivière, Roberto Baldoni, Harry Li, and José Pereira. Compositional gossip: a conceptual architecture for designing gossip-based applications. *ACM SIGOPS Operating Systems Review*, 41(5):43–50, 2007. (Cited on page 42.)

Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, 2001. (Cited on pages 12 and 13.)

Hans Sagan. *Space-filling curves*. Springer-Verlag New York, 1994. (Cited on page 57.)

Stefan Saroiu, P Krishna Gummadi, and Steven D Gribble. Measurement study of peer-to-peer file sharing systems. *Multimedia Computing and Networking (MMCN)*, 2002. (Cited on page 24.)

Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 1990a. (Cited on page 10.)

Mahadev Satyanarayanan. A survey of distributed file systems. *Annual Review of Computer Science*, 4(1):73–104, 1990b. (Cited on page 9.)

Bianca Schroeder and Garth A Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, 7:1–16, 2007. (Cited on page 2.)

Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing*, 7(4):337–350, 2010. (Cited on page 2.)

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009. (Cited on page 2.)

Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, 2003. (Cited on pages 3, 12, and 13.)

Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. (Cited on page 1.)

Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS*, 29(5), 1995. (Cited on page 10.)

Ricardo Vilaça, Francisco Cruz, and Rui Oliveira. On the expressiveness and trade-offs of large scale tuple stores. *On the Move to Meaningful Internet Systems, OTM 2010*, pages 727–744, 2010. (Cited on pages 12 and 13.)

Ricardo Vilaça, Rui Oliveira, and Jose Pereira. A correlation-aware data placement strategy for key-value stores. *Distributed Applications and Interoperable Systems*, pages 214–227, 2011. (Cited on page 57.)

Spyros Voulgaris and Maarten Steen. Epidemic-style management of semantic overlays for content-based searching. *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, 2005. (Cited on page 15.)

Spyros Voulgaris and Maarten van Steen. Vicinity: A pinch of randomness brings out the structure. *Middleware*, pages 21–40, 2013. (Cited on pages 15 and 70.)

Spyros Voulgaris, Daniela Gavidia, and Maarten Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005a. (Cited on pages 14, 15, 23, 30, 43, 54, 60, and 77.)

Spyros Voulgaris, Márk Jelasity, and Maarten Van Steen. A robust and scalable peer-to-peer gossiping protocol. *Agents and Peer-to-Peer Computing*, pages 47–58, 2005b. (Cited on pages 15 and 60.)

Feng Wang, Yongqiang Xiong, and Jiangchuan Liu. mtreebone: A collaborative tree-mesh overlay network for multicast video streaming. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):379–392, 2010. (Cited on pages 23 and 24.)

Liang Wang and Jussi Kangasharju. Measuring large-scale distributed systems: case of bittorrent mainline dht. *IEEE Thirteenth International Conference on Peer-to-Peer Computing*, pages 1–10, 2013. (Cited on pages 14 and 83.)

MyungKeun Yoon. Aging bloom filter with two active buffers for dynamic sets. *Knowledge and Data Engineering, IEEE Transactions on*, 22(1):134–138, 2010. (Cited on pages 40, 41, 47, and 48.)

B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Journal on Selected Areas in Communications*, 2006. (Cited on pages 12 and 13.)